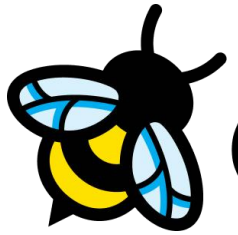
Linux Tracing System浅析

&



eBPF 框架开发经验分享

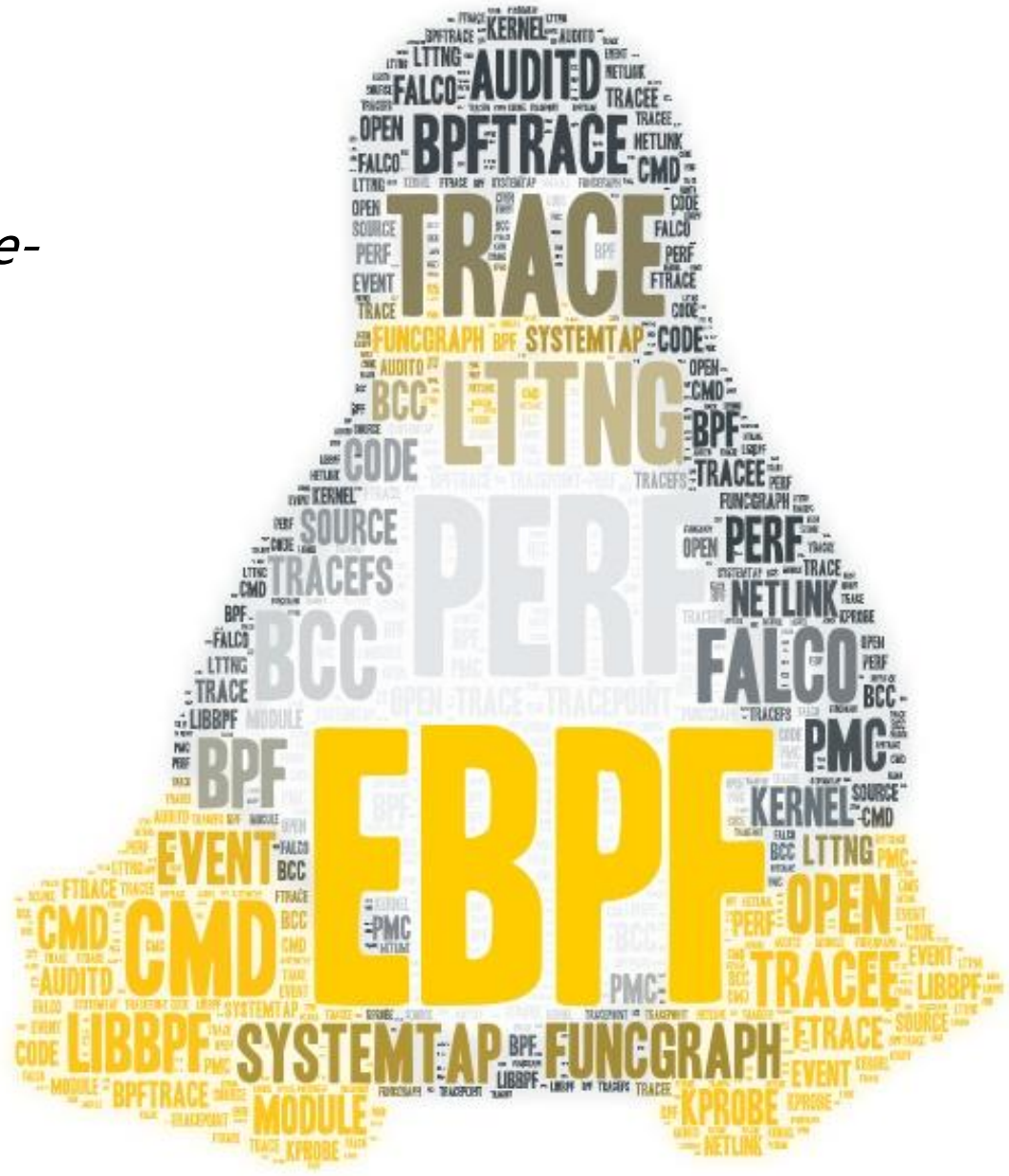
杨润青

rainkin1993 at gmail.com

Linux Tracing System浅析

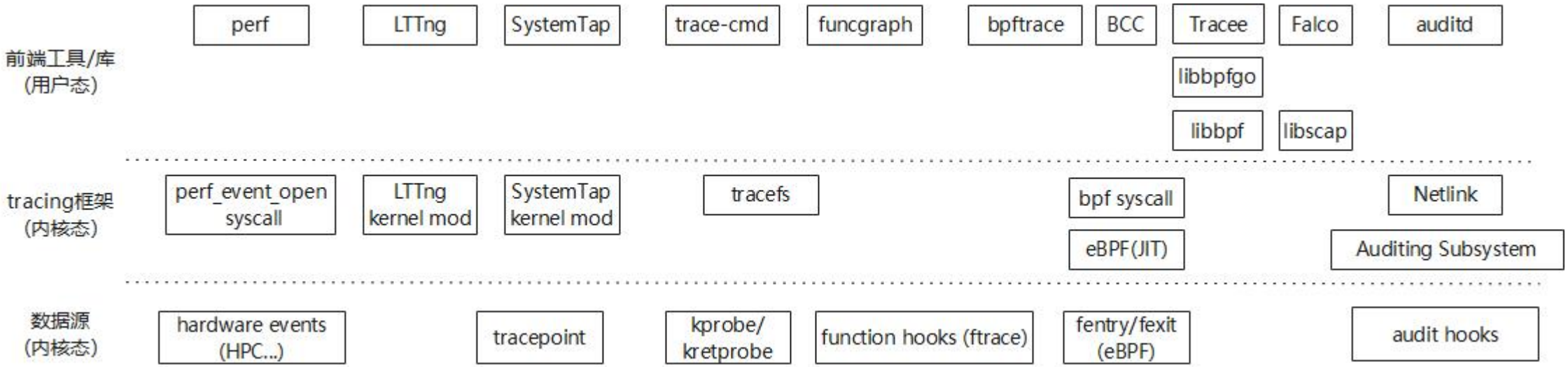
“Kprobe，Tracepoint，Linux Auditing subsystem(auditd)，SystemTap，LTTng，perf，trace-cmd，eBPF，bpftrace，BCC … ”

- 这些专业词语是什么意思?
- 它们之间有什么关系?
- 每种tracing技术的优缺点是什么?
- 应该选择哪种技术?
- 为什么eBPF会从中脱颖而出，近年来得到广泛关注?



Linux Tracing System

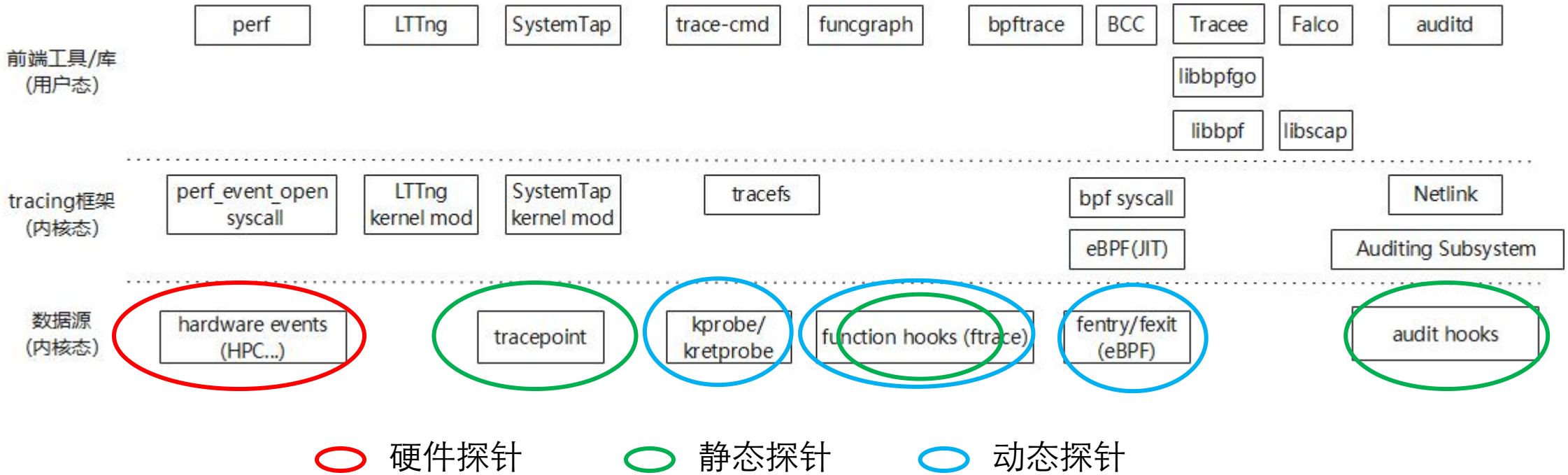
- **数据源**：提供数据的来源
- **Tracing 内核框架**：负责对接数据源，采集解析发送数据，并对用户态提供接口
- **前端工具/库**：对接Tracing内核框架，直接与用户交互，负责采集配置和数据分析



* Figure inspired by https://static.sched.com/hosted_files/osseu19/5f/unified-tracing-platform-oss-eu-2019.pdf

Linux Tracing System

- **数据源**：提供数据的来源
- **Tracing 内核框架**：负责对接数据源，采集解析发送数据，并对用户态提供接口
- **前端工具/库**：对接Tracing内核框架，直接与用户交互，负责采集配置和数据分析



* inspired by https://static.sched.com/hosted_files/osseu19/5f/unified-tracing-platform-oss-eu-2019.pdf

硬件探针(HPC)

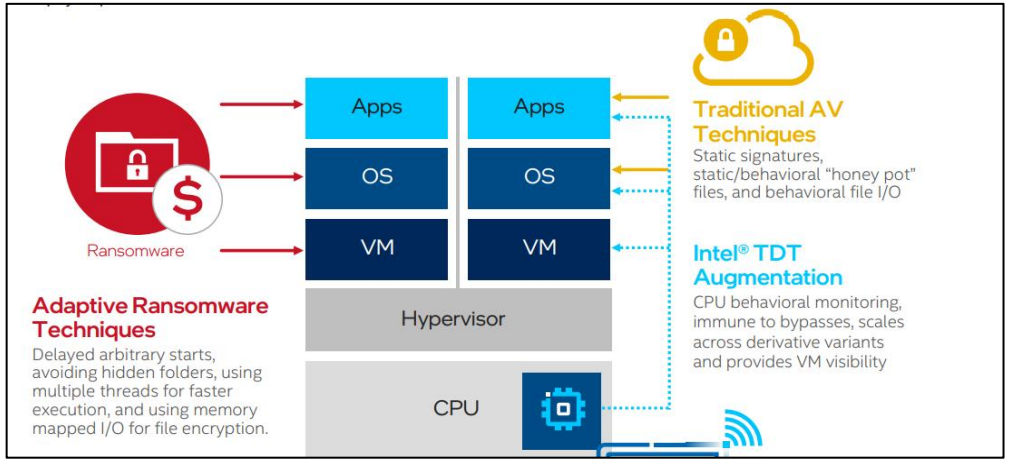
Hardware Performance Counter (HPC) 是CPU硬件提供的功能，它能够监控CPU级别的事件，比如执行的指令数，跳转指令数，Cache Miss等等,被广泛应用于性能调试（Vtune, Perf）、攻击检测等等。

HPC列表

Architectural Events	Description	Non-architectural Events	Description
1. Ins	Instruction retired	9. Uops_Retired	All micro-operations that retired
2. Clk	Unhalted core cycles	10. Mem_Load_Uops_Retired	Retired load uops
3. Br	Branch instructions	11. Mem_Store_Uops_Retired	Retired store uops
4. Arith_Ins	Arithmetic instructions	12. Br_Miss_Pred_Retired	Mispredicted branches that retired
5. Call	Near call instructions	13. Ret_Miss	Mispredicted return instructions
6. Call_D	Direct near call instructions	14. Call_D_Miss	Mispredicted direct call instructions
7. Call_ID	Indirect near call instructions	15. Br_Far	Far branches retired
8. Ret	Near return instructions	16. Br_Inst_Exec	Branch instructions executed
		17. ITLB_Miss	Misses in ITLB
		18. DTLB_Store_Miss	Store uops with DTLB miss
		19. DTLB_Load_Miss	Load uops with DTLB miss
		20. LLC_Miss	Longest latency cache miss

* Figure from S&P 19 SoK: The Challenges, Pitfalls, and Perils of Using Hardware Performance Counters for Security

Intel TDT



* Figure from Intel TDT

HPC数据案例

```
# perf stat -e cycles,instructions,branches gzip file1

 Performance counter stats for 'gzip file1':

     5,649,595,479 cycles                    #    2.942 GHz                      [83.43%]
     8,625,207,199 instructions              #    1.53  insns per cycle
                                             #    0.21  stalled cycles per insn [83.51%]
     1,488,797,176 branches                  #  775.351 M/sec                    [82.58%]

       1.936842598 seconds time elapsed
```

硬件探针(LBR)

Last Branch Record（LBR）是CPU硬件提供的另一种特性，它能够记录了每条分支(跳转)指令的源地址和目的地址。基于LBR硬件特性，可实现调用栈信息的记录。

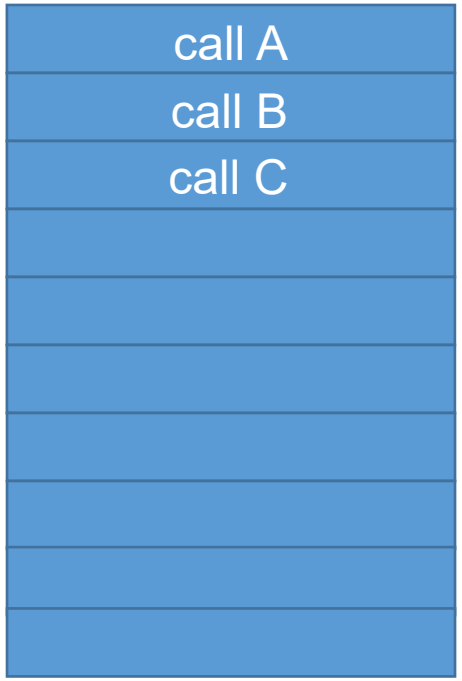
Source Code

```
1  #include <stdio.h>
2
3  void C(){
4      printf("%s", "C");
5  }
6
7  void B(){
8      printf("%s", "B");
9      C();
10 }
11
12 void A(){
13     printf("%s", "A");
14     B();
15 }
16
17 int main()
18 {
19     A();
20 }
21
```

Assembly Code

```
line19: call A
line13: printf(“A”)
line14: call B
line08: printf(“B”)
line09: call C
line04: printf(“C”)
line05: return
line10: return
line15: return
```

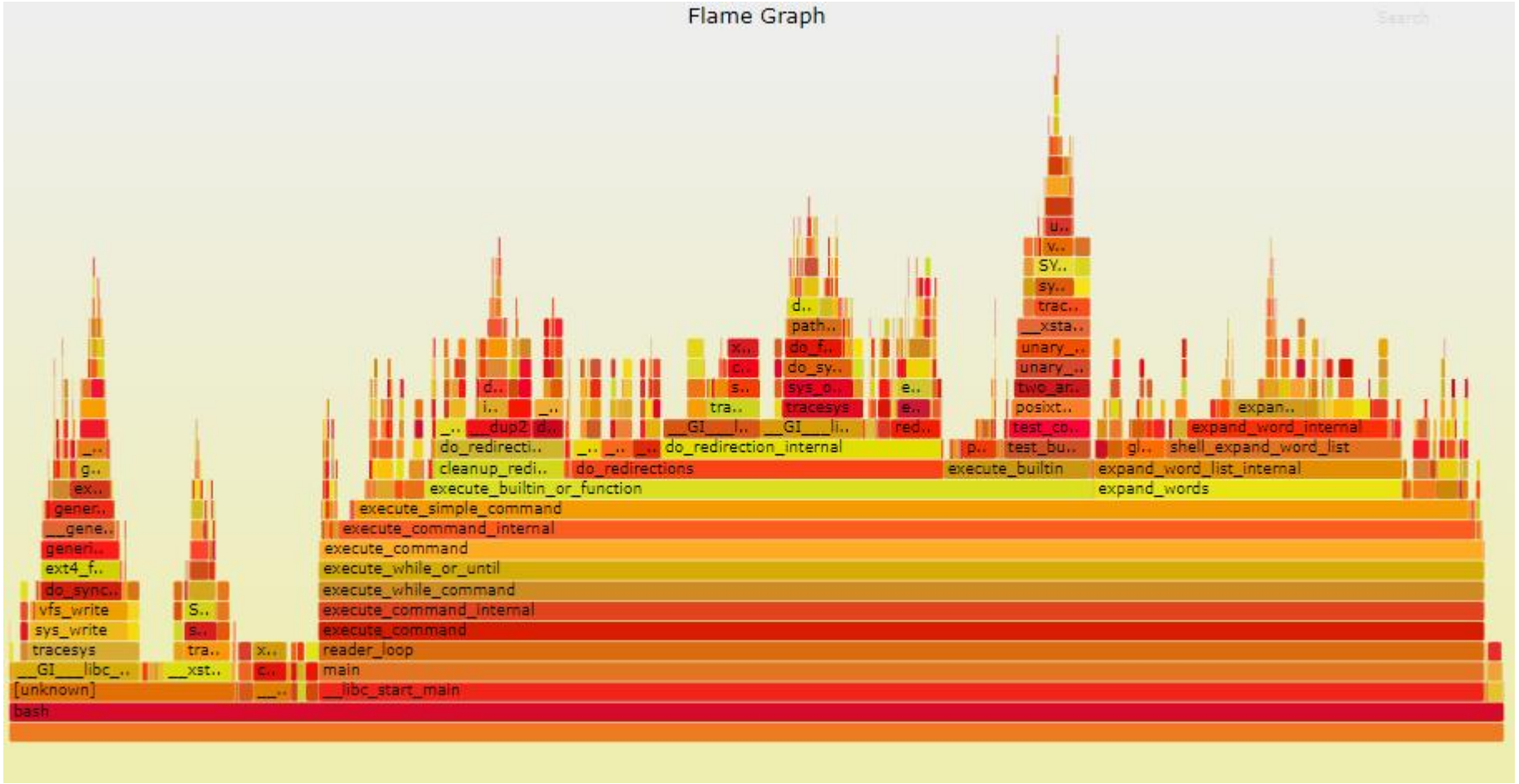
LBR



硬件探针(LBR)

我们调试程序时常使用的火焰图（Flame Graph）也可以基于LBR的数据生成

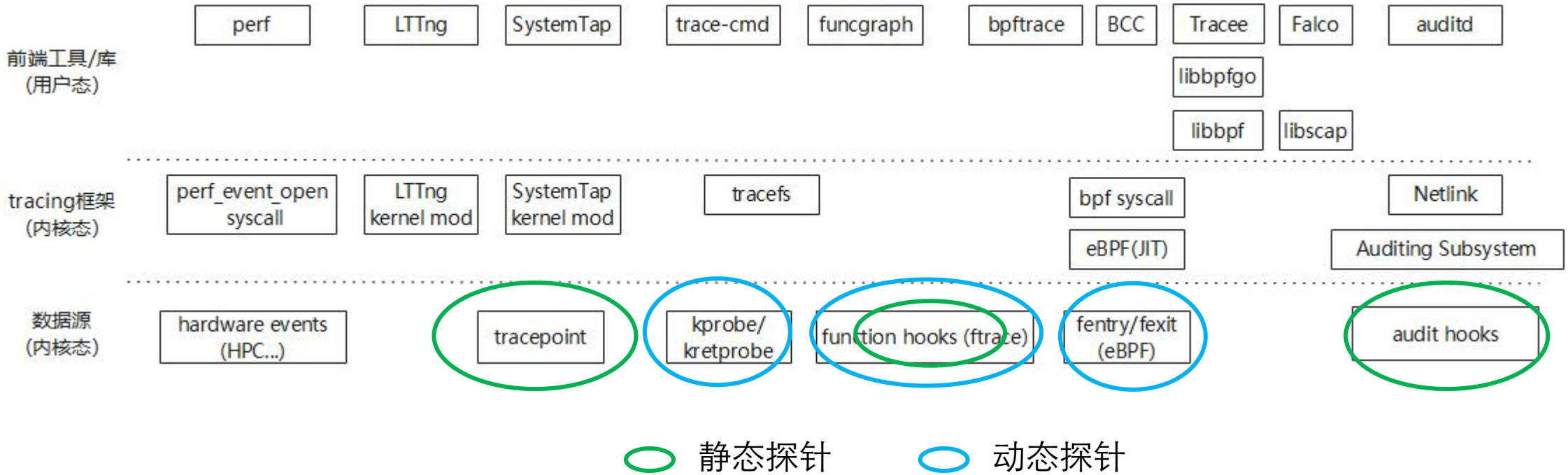
perf record -F 99 -a --call-graph lbr



* Figure from https://github.com/brendangregg/FlameGraph/tree/master

Linux Tracing System

- **数据源**：提供数据的来源
- **Tracing 内核框架**：负责对接数据源，采集解析发送数据，并对用户态提供接口
- **前端工具/库**：对接Tracing内核框架，直接与用户交互，负责采集配置和数据分析



* inspired by https://static.sched.com/hosted_files/osseu19/5f/unified-tracing-platform-oss-eu-2019.pdf

静态探针 VS 动态探针

- 通过静态探针（tracepoint sched_process_exec）监控进程执行二进制文件的行为

```
# trace-cmd record -e sched_process_exec
# trace-cmd report
      ps-184478 [000] 154243.947402: sched_process_exec:   filename=/usr/bin/ps pid=184478 old_pid=184478
    grep-184479 [002] 154243.947596: sched_process_exec:   filename=/usr/bin/egrep pid=184479 old_pid=184479
    grep-184480 [003] 154243.947681: sched_process_exec:   filename=/usr/bin/grep pid=184480 old_pid=184480
     awk-184481 [006] 154243.947783: sched_process_exec:   filename=/usr/bin/awk pid=184481 old_pid=184481
    grep-184479 [005] 154243.948385: sched_process_exec:   filename=/usr/bin/grep pid=184479 old_pid=184479
     cat-184482 [003] 154244.970658: sched_process_exec:   filename=/usr/bin/cat pid=184482 old_pid=184482
```

- 通过动态探针（kprobe exec_binprm）监控进程执行二进制文件的行为

```
# cat trace_exec.bt
#include <linux/binfmts.h>
kprobe:exec_binprm {
    $binprm = (struct linux_binprm *)arg0;
    printf("%s- %d: [%03d] %d: exec_binprm: filename=%s pid=%d old_pid=%d\n", comm, pid, cpu, nsecs/1000000000, str($binprm->filename), pid, pid);
}

# bpftrace trace_exec.bt
    sh-184478:  [000] 154247: exec_binprm: filename=/usr/bin/ps pid=184478 old_pid=184478
    sh-184479:  [002] 154247: exec_binprm: filename=/usr/bin/egrep pid=184479 old_pid=184479
    sh-184480:  [003] 154247: exec_binprm: filename=/usr/bin/grep pid=184480 old_pid=184480
    sh-184481:  [006] 154247: exec_binprm: filename=/usr/bin/awk pid=184481 old_pid=184481
 egrep-184479:  [005] 154247: exec_binprm: filename=/usr/bin/grep pid=184479 old_pid=184479
 bash-184482:   [003] 154248: exec_binprm: filename=/usr/bin/cat pid=184482 old_pid=184482
```

静态探针 VS 动态探针

2012年静态探针sched_process_exec被引入内核 [1]

```
TRACE_EVENT(sched_process_exec,
    TP_PROTO(struct task_struct *p, pid_t old_pid,
         struct linux_binprm *bprm),
    ...
);
```

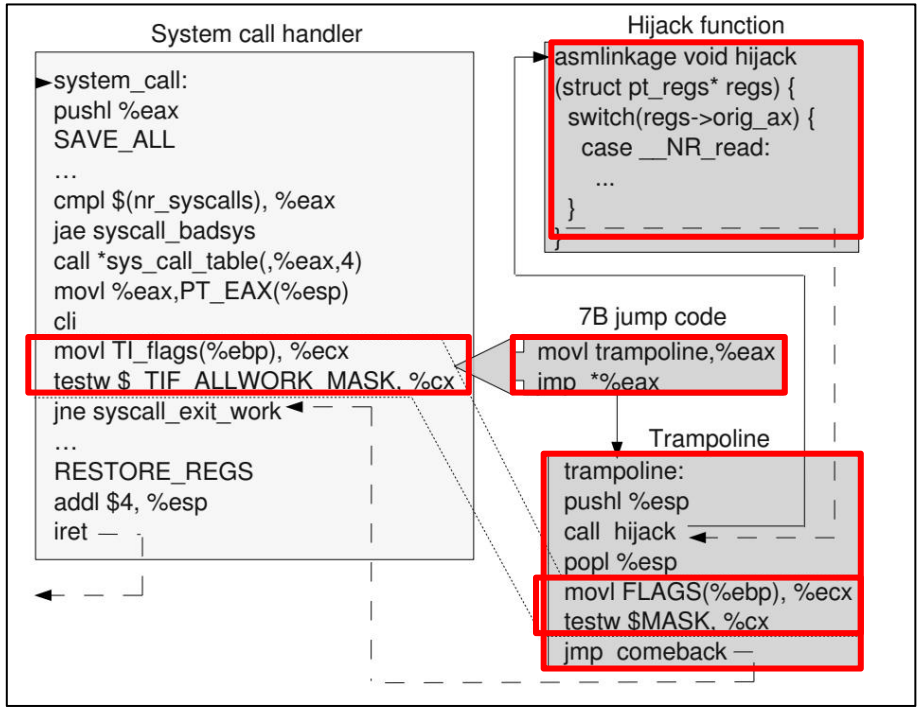
```
static int exec_binprm(struct linux_binprm *bprm)
{
    /* ... */


trace_sched_process_exec(current, old_pid, bprm);

    return 0;
}
```

[1] https://github.com/torvalds/linux/commit/4ff16c25e2cc48cbe6956e356c38a25ac063a64d

典型代表：Kernel Tracepoint

- 稳定（内核开发者会负责维护该函数的稳定性）
- 性能好
- 需要修改内核代码来添加新的静态探针
- 内核支持的静态探针数量有限

动态探针示意图



典型代表：Kprobe

- 可以Hook几乎所有的内核函数
- 不稳定（函数的变更可能导致程序失效）
- 性能相对较差

* photo from https://core.ac.uk/download/pdf/62916134.pdf

优先使用静态Hook (如Kernel Tracepoint)

Compile time hooks + Dynamic Function Tracer(ftrace)

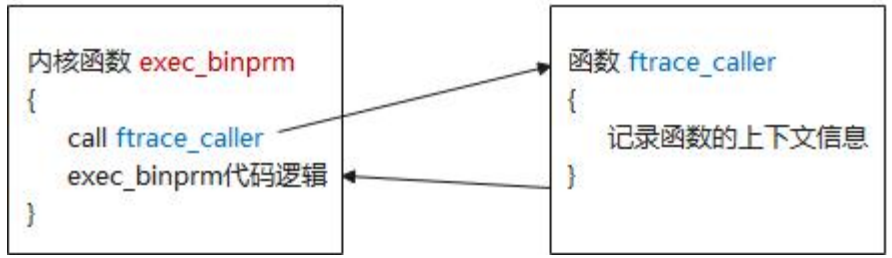
```
# trace-cmd record -p function -l __audit_inode --func-stack
# trace-cmd report
        bash-95922 [004] 76378.594743: function:       __audit_inode
        bash-
95922 [004] 76378.594744: kernel_stack:   <stack trace >
        => __audit_inode (ffffffff8f97e6e5)
        => do_open (ffffffff8fb0e69d)
        => path_openat (ffffffff8fb10f9a)
        => do_filp_open (ffffffff8fb1336c)
        => do_open_execat (ffffffff8fb070f1)
        => open_exec (ffffffff8fb0725c)
        => load_elf_binary (ffffffff8fb8b056)
        => search_binary_handler (ffffffff8fb07541)
        => exec_binprm (ffffffff8fb07721)
        => __do_execve_file (ffffffff8fb082c1)
        => do_execve (ffffffff8fb08e47)
        => call_usermodehelper_exec_async (ffffffff8f8bb84b)
        => ret_from_fork (ffffffff8f8044b2)
```

Compile time hooks + Dynamic Function Tracer(ftrace)

- 内核编译未开启Ftrace功能

- 内核编译开启Ftrace功能
- 未开启function tracer

- 内核编译开启Ftrace功能
- 开启function tracer，跟踪内核函数exec_binprm



Compile time hooks + Dynamic Funcgraph Tracer(ftrace)

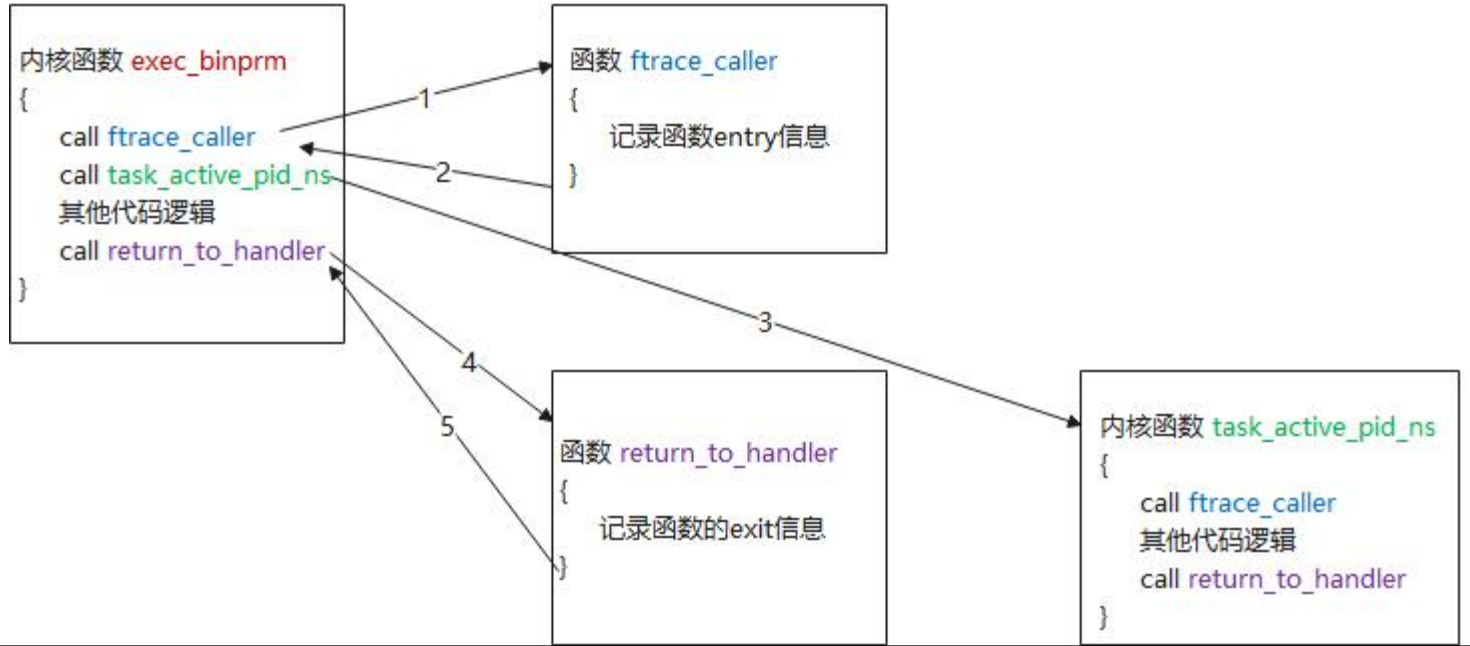
```
# ./funcgraph -m 3 exec_binprm
Tracing "exec_binprm"... Ctrl-C to end.
  4)               |  exec_binprm() {
  4)   0.632 us    |    task_active_pid_ns();
  4)   0.251 us    |    __task_pid_nr_ns();
  4)               |    search_binary_handler() {
  4)   6.094 us    |      kernel_read();
  4)   1.384 us    |      security_bprm_check();
  4)   0.337 us    |      _raw_read_lock();
  4)   0.358 us    |      try_module_get();
  4)   0.260 us    |      _raw_read_lock();
  4)   0.209 us    |      module_put();
  4)   0.180 us    |      try_module_get();
  4)   0.170 us    |      load_script();
  4)   0.174 us    |      _raw_read_lock();
  4)   0.183 us    |      module_put();
  4)   0.173 us    |      try_module_get();
  4) ! 485.507 us  |      load_elf_binary();
  4)   0.247 us    |      _raw_read_lock();
  4)   0.203 us    |      module_put();
  4) ! 502.488 us  |    }
  4)   0.308 us    |    proc_exec_connector();
  4) ! 506.758 us  |  }
```

Compile time hooks + Dynamic Funcgraph Tracer(ftrace)

- 内核编译未开启Ftrace功能

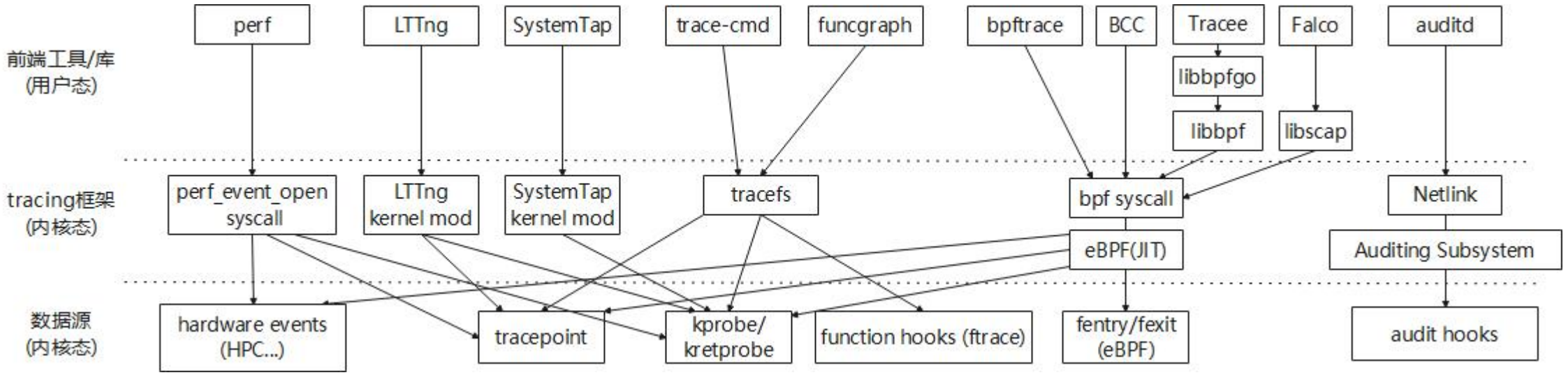
- 内核编译开启Ftrace功能
- 未开启function graph tracer

- 内核编译开启Ftrace功能
- 开启function graph tracer，跟踪内核函数exec_binprm



Linux Tracing System

- **数据源**：提供数据的来源
- **Tracing 内核框架**：负责对接数据源，采集解析发送数据，并对用户态提供接口
- **前端工具/库**：对接Tracing内核框架，直接与用户交互，负责采集配置和数据分析



* Figure inspired by https://static.sched.com/hosted_files/osseu19/5f/unified-tracing-platform-oss-eu-2019.pdf

备受争议的audit hooks（Linux auditing subsystem）

- Audit hooks是独立于kprobe、tracepoint之外的数据源，它通过在syscall和文件操作等内核源代码上插入自定义的hook函数来实现syscall和文件行为的监控。
- 相比其他技术，性能非常差。

“**In summary, the code is a giant mess**. The way it works is nearly incomprehensible. It contains at least one severe bug. I‘d love to see it fixed, but for now, distributions seem to think that enabling CONFIG_AUDITSYSCALL is a reasonable thing to do, and I’d argue that it‘s actually a terrible choice for anyone who doesn’t actually need syscall audit rules. And I don‘t know who needs these things. ”

—— 2014年，LWN: Who audits the audit code?

Linux Tracing System 发展历程

- **2004年4月，**Linux Auditing subsystem(auditd)被引入内核2.6.6-rc1
- **2005年4月**，Kprobe被引入内核2.6.11.7
- **2006年**，LTTng发布（至今没有合入内核）
- **2008年10月**，Kernel Tracepoint 被引入内核（v2.6.28）。
- **2008年**，Ftrace被引入内核（包括compile time function hooks）。
- **2009年**，perf被引入内核
- **2009年**，SystemTap发布（至今没有合入内核）
- **2014年，**Alexei Starovoitov将eBPF引入内核

Tracing内核框架对比

采集框架	稳定性	Linux内核内置	内核编程	性能
Linux Auditing Subsystem	🟢	🟢	🔴	🔴
SystemTap	🔴	🔴	🟢	🟢
LTTng	🔴	🔴	🔴	🟢
perf/ftrace	🟢	🟢	🔴	🟢
eBPF	🟢	🟢	🟢	🟢

eBPF的优势：

- **稳定**：通过验证器，**防止用户编写的程序导致内核崩溃**。
  - 相比内核模块，eBPF的稳定性更容易被产品线接受。
- **免安装**：eBPF**内置于Linux内核**，无需安装额外依赖，开箱即用。
- **内核编程**：支持开发者插入**自定义的代码逻辑**（包括数据采集、分析和过滤）到内核中运行

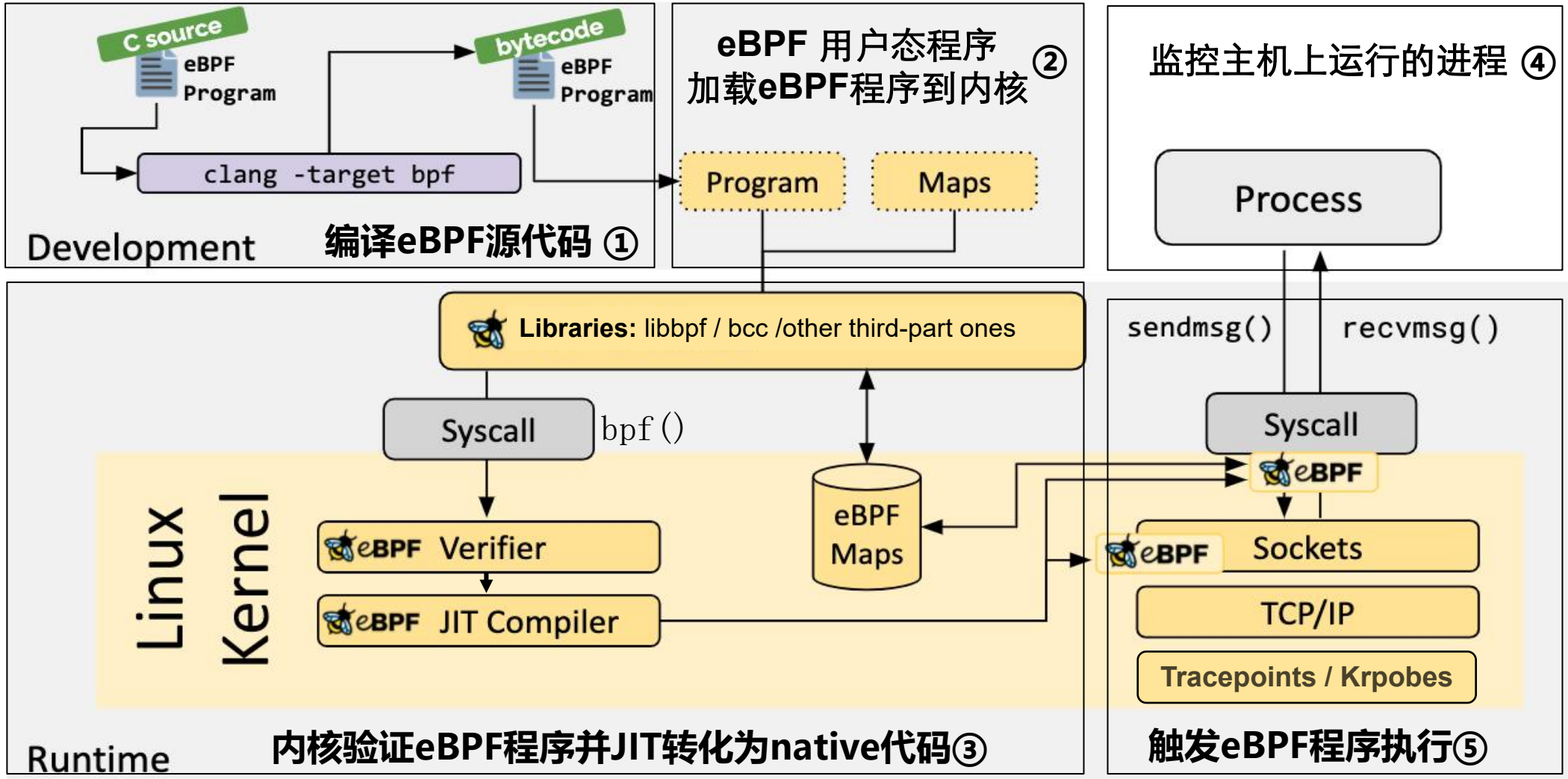
eBPF框架开发经验分享

eBPF目前正在高速发展，很多坑和解决办法缺乏官方文档。本次讲座主要介绍本人在eBPF开发实践中经常遇到的问题，包括

1. eBPF开发框架的选择
2. eBPF如何跨内核版本运行（可移植性）
3. 如何为低版本内核生成BTF文件
4. eBPF验证机制与编译器优化机制的不一致问题
5. eBPF在ARM架构遇到的问题
6. ...

eBPF背景介绍—工作流程

eBPF程序分为两部分: 用户态和内核态代码。



* Photo modified from ebpf.io

初学者常见疑问：编写eBPF工具是否一定要使用C语言?

eBPF程序分为两部分: 用户态和内核态代码。

eBPF内核代码:

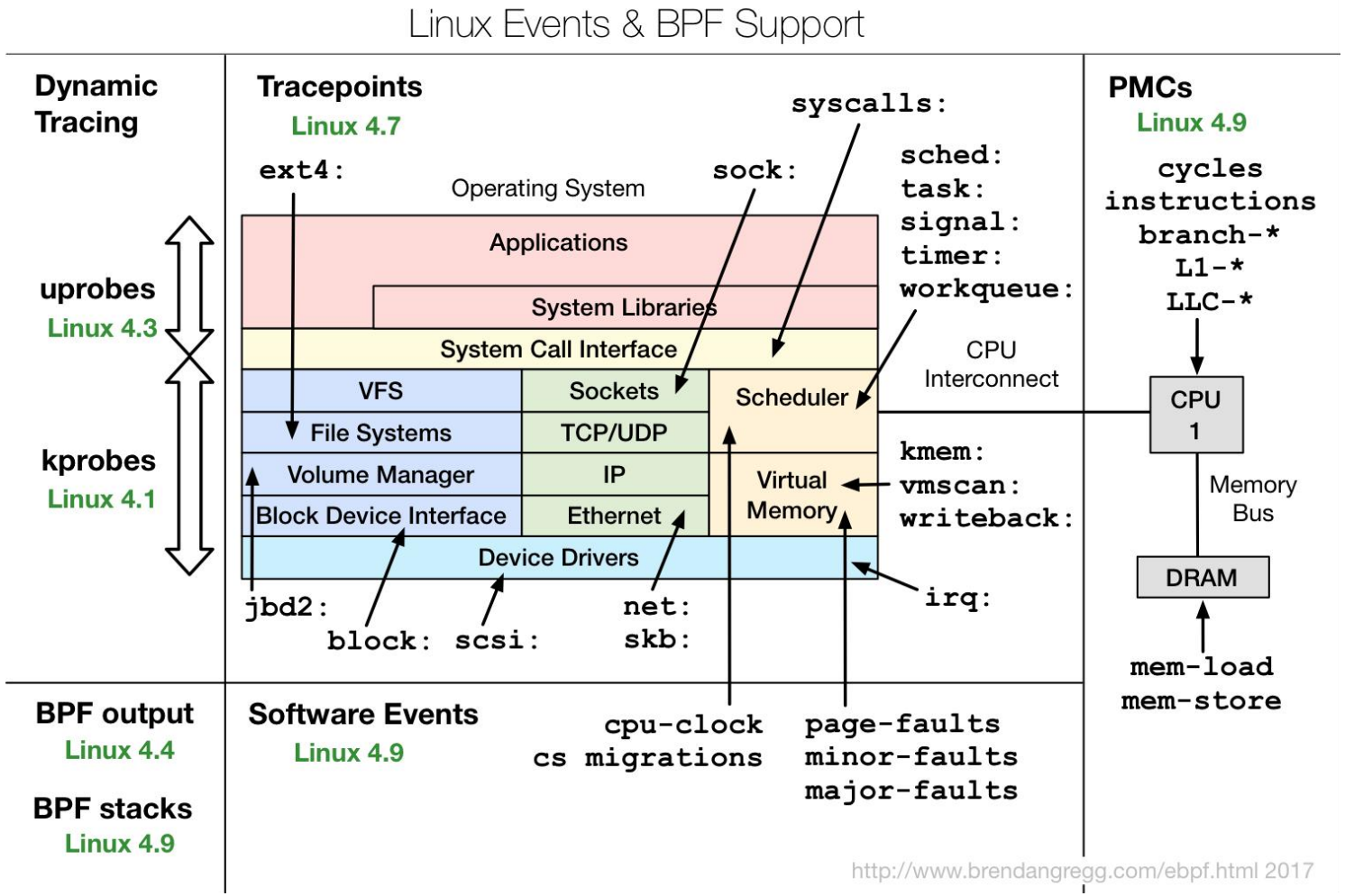
- 这个代码首先需要经过编译器（比如LLVM）编译成eBPF字节码，然后字节码会被加载到内核执行。所以这部分代码理论上用什么语言编写都可以，只要编译器支持将该语言编译为eBPF字节码即可。
- 目前绝大多数工具都是用的C语言来编写eBPF内核代码，包括BCC。
- bpftrace提供了一种易用的脚本语言来帮助用户快速高效的使用eBPF功能，其背后的原理还是利用LLVM将脚本转为eBPF字节码。

eBPF用户态代码:

这个代码负责将eBPF内核程序加载到内核，与eBPF MAP交互，以及接收eBPF内核程序发送出来的数据。

上述功能本质上是通过Linux OS提供的syscall（bpf syscall + perf_event_open syscall）完成的，因此这部分代码你可以用任何语言实现。比如BCC使用python，libbpf使用c或者c++，TRACEE使用Go等等。

eBPF背景介绍—数据源



数据源：

- (Raw) Tracepoints
- Fentry/Fexit
- USDT
- Kprobes/Uprobes
- LSM (允许拦截用户行为），Linux5.7
- PMCs
- …

* photo from http://www.brendangregg.com/ebpf.html

eBPF开发框架的发展历程

- **2014年9月 引入了bpf() syscall，将eBPF引入用户态空间。**
  - 其中自带了一个迷你libbpf库，简单对bpf()进行了封装，功能是将**eBPF字节码**加载到内核。
  - **2015年2月份** Kernel 3.19 引入bpf_load.c/h文件，对上述迷你libbpf库再进行封装，功能是将**eBPF elf二进制文件**加载到内核（目前已过时，不建议使用）。
- **2015年4月 BCC项目创建，提供了eBPF一站式编程。**
  - 创建之初，基于上述**迷你libbpf库**来加载eBPF字节码。
  - 提供了Python接口。
- **2015年11月 Kernel 4.3 引入标准库 libbpf**
  - 该标准库由Huawei 2012 OS内核实验室的王楠提交。
- **2018年 为解决BCC的缺陷，CO-RE（Compile Once， Run Everywhere）的想法被提出并实现**
  - **达成共识：libbpf + BTF + CO-RE代表了eBPF的未来**，BCC底层实现逐步转向libbpf。

eBPF程序如何跨内核版本运行（可移植性）

- **eBPF程序移植性差：**
  - 在内核版本A上编译的eBPF程序，无法直接在另外一个内核版本B上运行。
  - 根本原因在于**eBPF程序访问的内核数据结构(内存空间）是不稳定的**，经常随内核版本变化而变化。
- **BCC通过在部署机器上动态编译eBPF源代码来解决移植性问题。**
  - 每一次eBPF程序运行都需要进行一次编译。
  - 需要在部署机器上按照上百兆大小的依赖：Clang/LLVM + Linux headers。
  - Clang/LLVM编译过程消耗大量资源（CPU/内存），对业务造成影响。

可移植性解决方案——Compile Once，Run Everywhere

CO-RE的实现原理：

- **BTF**
  - 将内核数据结构信息高效压缩和存储（相比于DWARF，可达到超过100倍的压缩比）
- **编译器支持**
  - 编译eBPF代码的时候记录下relocation相关的信息
- **eBPF开发框架的支持（libbpf）**
  - 基于BTF和编译器提供的信息，动态relocate数据结构
- **内核**
  - 内核几乎不需要改动！

内核5.2+版本才自带BTF文件，低版本内核缺少BTF文件来支持CO-RE特性

内核BTF文件生成流程

```
# generate .BTF typeinfo from DWARF debuginfo
# ${1} - vmlinux image
# ${2} - file to dump raw BTF data into
gen_btf()
{
    // 检查软件pahole是否可用(pahole版本>=1.16)

    ...

    //  利用pahole软件在vmlinux中生成BTF信息
    //  pahole -J, --btf_encode
    //        Encode  BTF information from DWARF, used in the Linux kernel build process when CONFIG_DEBUG_INFO_BTF=y is present,
    //        introduced in Linux v5.2. Used to implement features such as BPF CO-RE (Compile Once - Run Everywhere).
    LLVM_OBJCOPY="${OBJCOPY}" ${PAHOLE} -J ${PAHOLE_FLAGS} ${1}

    // 将BTF信息单独输出到BTF文件${2}中
    ${OBJCOPY} --only-section=.BTF --set-section-flags .BTF=alloc,readonly \
        --strip-all ${1} ${2} 2>/dev/null

    ...
}
```

- BTF文件的生成并不需要改动内核！只依赖：
  - 带有debug info的vmlinux image
  - pahole 软件
  - LLVM 软件

在不需要打内核补丁的情况下，为低版本内核生成BTF文件理论可行！

为低版本内核生成BTF文件

准备工作：

- 安装pahole软件（1.16+）
  - https://git.kernel.org/pub/scm/devel/pahole/pahole.git
- 安装LLVM（11+）
- 获取目标低版本内核的vmlinux文件（带有debug info），文件保存在{vmlinux_file_path}
  - 通过源下载
    - 比如对于CentOS，通过yum install kernel-debuginfo可以下载vmlinux
  - 源码编译内核，获取vmlinux

生成BTF：

- 利用pahole在vmlinux文件中生成BTF信息，执行以下命令：
  - pahole -J {vmlinux_file_path}
- 将BTF信息单独输出到新文件{BTF_file_path}，执行以下命令：
  - llvm-objcopy --only-section=.BTF --set-section-flags .BTF=alloc,readonly --strip-all {vmlinux_file_path} {BTF_file_path}
- 去除非必要的符号信息，降低BTF文件的大小，得到最终的BTF文件（大小约2~3MB）：
  - strip -x {BTF_file_path}

一个print引发的惨案

```
SEC("kprobe/do_unlinkat")
int BPF_KPROBE(do_unlinkat, int dfd, struct filename *name)
{
        // 获取一个数组指针array（数组MAX_SIZE为16个字节）
        u32 key = 0;
        char *array = bpf_map_lookup_elem(&array_map, &key);
        if (array == NULL)
                return 0;
        // 获取当前运行程序的CPU编号(当前机器的CPU有16个核)
        unsigned int pos = bpf_get_smp_processor_id();;
        // 修改数值
        if (pos < MAX_SIZE){
                array[pos] = 1;
                pos += 1;
        }
        // debug代码，输出一些上下文信息
        bpf_printk("debug %d %d %d\n", bpf_get_current_pid_tgid() >> 32,
                   bpf_get_current_pid_tgid(), array[1]);
        // 修改数值
        if (pos < MAX_SIZE)
                array[pos] = 1;
        return 0;
}
```

代码测试环境均为：
1. LLVM 11
2. Ubuntu 5.8.0内核版本
3. libbpf commit @9c44c8a

一个print引发的惨案

```
SEC("kprobe/do_unlinkat")
int BPF_KPROBE(do_unlinkat, int dfd, struct filename *name)
{
        // 获取一个数组指针array（数组MAX_SIZE为16个字节）
        u32 key = 0;
        char *array = bpf_map_lookup_elem(&array_map, &key);
        if (array == NULL)
                return 0;
        // 获取当前运行程序的CPU编号(当前机器的CPU有16个核)
        unsigned int pos = bpf_get_smp_processor_id();;
        // 修改数值
        if (pos < MAX_SIZE){
                array[pos] = 1;
                pos += 1;
        }



        // 修改数值
        if (pos < MAX_SIZE)
                array[pos] = 1;
        return 0;
}
```

无法通过验证
因为R0寄存器(变量pos)没有进行边界检查

仅仅删除源码中的print代码，导致无法通过验证

eBPF验证器——边界检查案例

- eBPF会被LLVM编译为eBPF字节码
- eBPF字节码需要通过eBPF verifier的(静态)验证后，才能真正运行
- 边界检查是eBPF verify的重点工作，防止eBPF程序内存越界访问

```
SEC("kprobe/do_unlinkat")
int BPF_KPROBE(do_unlinkat, int dfd, struct filename *name)
{
    // 获取一个数组指针array（数组MAX_SIZE为16个字节）
    u32 key = 0;
    char *array = bpf_map_lookup_elem(&array_map, &key);
    if (array == NULL)
            return 0;
    // 获取当前运行程序的CPU编号(当前机器的CPU有16个核)
    unsigned int pos = bpf_get_smp_processor_id();;
    // 根据下表修改数组的值
    array[pos] = 1;
    return 0;
}
```

eBPF验证器——边界检查案例

```
0000000000000000 <do_unlinkat>:
; int BPF_KPROBE(do_unlinkat, int dfd, struct filename *name)
       0:       r1 = 0
;        u32 key = 0;
       1:       *(u32 *)(r10 - 4) = r1
       2:       r2 = r10
       3:       r2 += -4
;          char *array = bpf_map_lookup_elem(&array_map, &key);
       4:       r1 = 0 ll
       6:       call 1
       7:       r6 = r0
;          if (array == NULL)
       8:       if r6 == 0 goto +6 <LBB0_2>
;        unsigned int pos = bpf_get_smp_processor_id();;
       9:       call 8
;                array[pos] = 1;
      10:       r0 <<= 32
      11:       r0 >>= 32
      12:       r6 += r0
      13:       r1 = 1
;                array[pos] = 1;
      14:       *(u8 *)(r6 + 0) = r1
```

14行无法通过验证

因为R6寄存器(变量pos)没有进行边界检查

eBPF验证器——边界检查案例

```
0000000000000000 <do_unlinkat>:
; int BPF_KPROBE(do_unlinkat, int dfd, struct filename *name)
       0:       r1 = 0
;        u32 key = 0;
       1:       *(u32 *)(r10 - 4) = r1
       2:       r2 = r10
       3:       r2 += -4
;          char *array = bpf_map_lookup_elem(&array_map, &key);
       4:       r1 = 0 ll
       6:       call 1
       7:       r6 = r0
;          if (array == NULL)
       8:       if r6 == 0 goto +7 <LBB0_3>
;        unsigned int pos = bpf_get_smp_processor_id();;
       9:       call 8
      10:       r0 <<= 32
      11:       r0 >>= 32
;        if (pos < MAX_SIZE)
      12:       if r0 > 15 goto +3 <LBB0_3>
;                  array[pos] = 1;
      13:       r6 += r0
      14:       r1 = 1
;                  array[pos] = 1;
      15:       *(u8 *)(r6 + 0) = r1
```

解决方案：添加边界检查代码

一个print引发的惨案——eBPF验证机制与编译器优化机制的不一致问题 (1)

```
SEC("kprobe/do_unlinkat")
int BPF_KPROBE(do_unlinkat, int dfd, struct filename *name)
{
        // 获取一个数组指针array（数组MAX_SIZE为16个字节）
        u32 key = 0;
        char *array = bpf_map_lookup_elem(&array_map, &key);
        if (array == NULL)
                return 0;
        // 获取当前运行程序的CPU编号(当前机器的CPU有16个核)
        unsigned int pos = bpf_get_smp_processor_id();;
        // 修改数值
        if (pos < MAX_SIZE){
                array[pos] = 1;
                pos += 1;
        }



        // 修改数值
        if (pos < MAX_SIZE)
                array[pos] = 1;
        return 0;
}
```

无法通过验证
因为R0寄存器(变量pos)没有进行边界检查

仅仅删除源码中的print代码，导致无法通过验证

一个print引发的惨案——eBPF验证机制与编译器优化机制的不一致问题 (1)

编译器存在自身的代码优化机制，逻辑相似的代码可能会生成完全不同的eBPF字节码

源代码

```
if (pos < MAX_SIZE){
    array[pos] = 1;
    pos += 1;
}
```

eBPF字节码

```
;       if (pos < MAX_SIZE){
      13:       if r1 > 15 goto +9 <LBB0_4>
;               array[pos] = 1;
      14:       r3 = r6
      15:       r3 += r1
      16:       r2 = 1
      17:       *(u8 *)(r3 + 0) = r2
```

通过验证，可以运行

对r1进行了边界检查，并用**r1**来访问了数组array

```
if (pos < MAX_SIZE)
    array[pos] = 1;
```

```
;       if (pos < MAX_SIZE) {
      18:       if r1 == 15 goto +4 <LBB0_4>
;               array[pos] = 1;
      19:       r0 <<= 32
      20:       r0 >>= 32
;               pos += 1;
      21:       r0 += r6
;               array[pos] = 1;
      22:       *(u8 *)(r0 + 1) = r2
```

无法通过验证，寄存器r0没有边界检查

对r1进行了边界检查，但却用**r0+1**来访问了数组array

根本原因：编译优化丢失了部分信息，导致eBPF verifier无法获得完整的上下文信息

一个print引发的惨案——eBPF验证机制与编译器优化机制的不一致问题 (2)

```c
SEC("kprobe/do_unlinkat")
int BPF_KPROBE(do_unlinkat, int dfd, struct filename *name)
{
    // 获取一个数组指针array（数组MAX_SIZE为16个字节）
    u32 key = 0;
    char *array = bpf_map_lookup_elem(&array_map, &key);
    if (array == NULL)
            return 0;
    // 获取当前运行程序的CPU编号(当前机器的CPU有16个核)
    unsigned long pos = bpf_get_smp_processor_id();;
    // 修改数值
    if (pos < MAX_SIZE){
        array[pos] = 1;
    }
    return 0;
}
```

一个print引发的惨案——eBPF验证机制与编译器优化机制的不一致问题 (2)

```c
SEC("kprobe/do_unlinkat")
int BPF_KPROBE(do_unlinkat, int dfd, struct filename *name)
{
    // 获取一个数组指针array（数组MAX_SIZE为16个字节）
    u32 key = 0;
    char *array = bpf_map_lookup_elem(&array_map, &key);
    if (array == NULL)
            return 0;
    // 获取当前运行程序的CPU编号(当前机器的CPU有16个核)
    unsigned long pos = bpf_get_smp_processor_id();;
    // 修改数值
    if (pos < MAX_SIZE){
        for (unsigned long i = 0; i < MAX_SIZE; i++)
            bpf_printk("debug %d %d %d\n", bpf_get_current_pid_tgid()
                        >> 32, bpf_get_current_pid_tgid(), array[i]);
        array[pos] = 1;
    }
    return 0;
}
```

无法通过验证
因为R1寄存器(变量pos)没有进行边界检查

仅仅在源码中添加了print代码，导致无法通过验证

一个print引发的惨案——eBPF验证机制与编译器优化机制的不一致问题 (2)

源代码

```
if (pos < MAX_SIZE)
    array[pos] = 1;
```

eBPF字节码

```
;       if (pos < MAX_SIZE){
      12:       if r0 > 15 goto +3 <LBB0_3>
;               array[pos] = 1;
      13:       r6 += r0
      14:       r1 = 1
;               array[pos] = 1;
      15:       *(u8 *)(r6 + 0) = r1
```

通过验证，可以运行

```
if (pos < MAX_SIZE){
   for (…)
      bpf_printk(…);
   array[pos] = 1;
}
```

```
;       if (pos < MAX_SIZE){
      12:       if r0 > 15 goto +24 <LBB0_5>
      13:       *(u64 *)(r10 - 16) = r0
          ...
;               array[pos] = 1;
      33:       r1 = *(u64 *)(r10 - 16)
      34:       r7 += r1
      35:       r1 = 1
      36:       *(u8 *)(r7 + 0) = r1
```

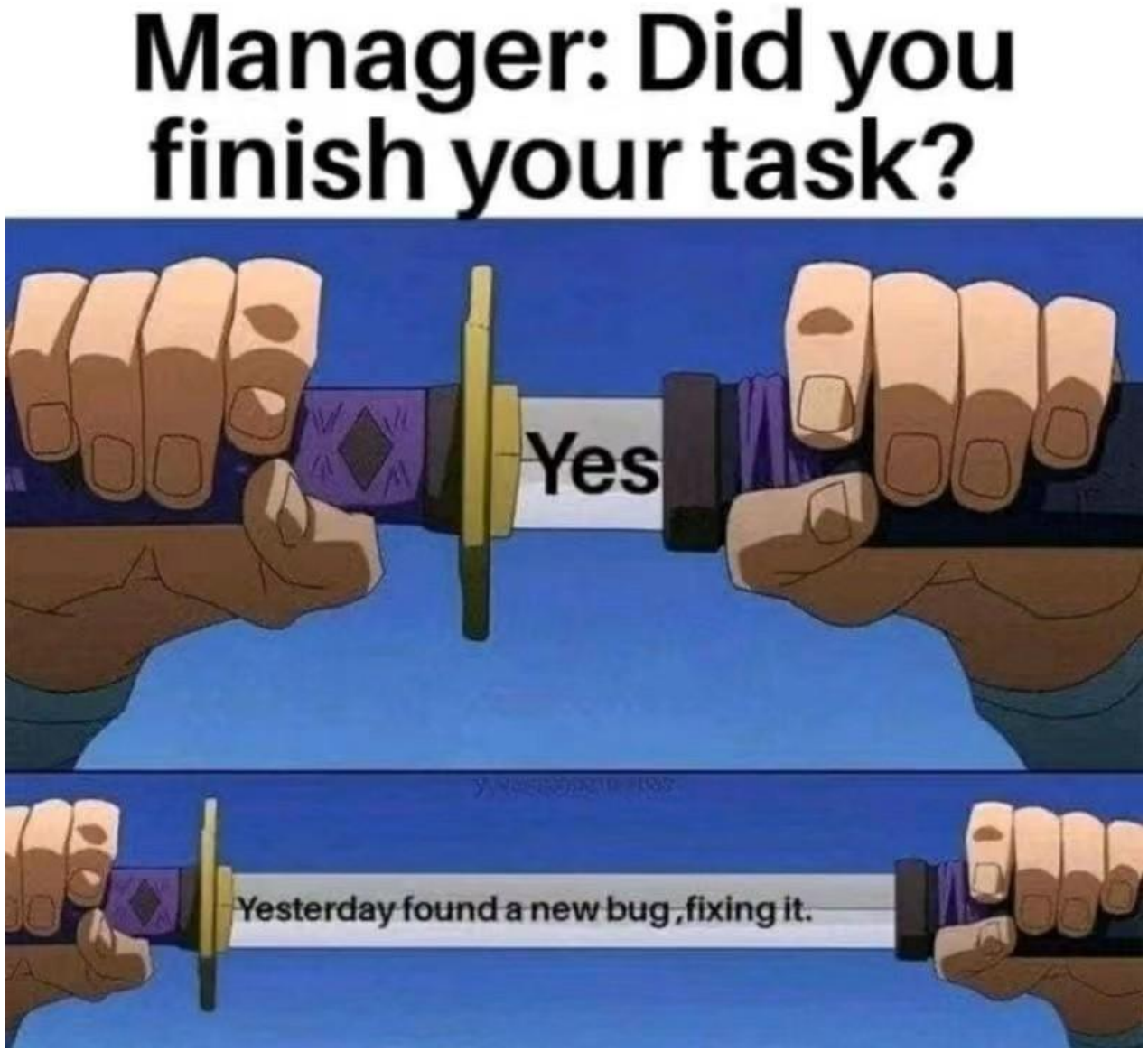
无法通过验证，
寄存器r1没有边界检查

根本原因：寄存器溢出/重加载后，状态丢失

eBPF验证机制与编译器优化机制的不一致问题 的缓解思路

1. 在源码中加入 &= 操作符，引导编译器生成理想的eBPF字节码
   - array[pos &= MAX_SIZE - 1] = 1;
2. 如果上述方法失效，无法引导编译器，那么针对出错的部分源代码人工编写eBPF字节码，替代编译器生成的字节码

```
#define STR(s) #s
#define XSTR(s) STR(s)
#define asm_variable_bound_check(variable)              \
({                                                      \
        asm volatile (                                  \
                "%[tmp] &= " XSTR(MAX_SIZE - 1) " \n" \
                :[tmp]"+&r"(variable)                   \
                );                                      \
})

asm_check(pos);
array[pos] = 1;
```


Manager: Did you finish your task?
Yes
Yesterday found a new bug, fixing it.

Q&A

技术选型—最低内核要求

功能	内核版本
Kprobes	4.1
Uprobes	4.3
Tracepoints	4.7
Timers/profiling	4.9
bpf2bpf calls	4.16
Kprobes + libbpf	4.17
Raw Tracepoints	4.17
Global variables	5.2
Bounded Loop	5.3
LSM Hook（拦截）	5.7
Ring buffers	5.8

内核编译选项要求

内核编译选项最低要求				
序号	功能	内核版本	内核编译选项	描述
1	基础	/	CONFIG_BPF_SYSCALL	启用bpf()系统调用。
2			CONFIG_BPF_JIT	BPF的过滤功能通常由一个解释器解释执行BPF虚拟机指令的方式工作。开启此项,内核在加载过滤指令后,会将其编译为本地指令,以加快执行速度。
3			CONFIG_HAVE_BPF_JIT	是CONFIG_BPF_JIT的依赖，适用于Linux内核版本4.1到4.6
4			CONFIG_HAVE_EBPF_JIT	是CONFIG_BPF_JIT的依赖，适用于Linux内核4.7版及更高版本
5			CONFIG_HAVE_CBPF_JIT	是CONFIG_BPF_JIT的依赖
6			CONFIG_MODULES	是CONFIG_BPF_JIT以及CONFIG_KPROBES的依赖
7			CONFIG_BPF	BPF VM 需要开启该选项
8			CONFIG_BPF_EVENTS	允许用户将BPF程序附加到kprobe、uprobe和tracepoint
9			CONFIG_PERF_EVENTS	启用内核对软件和硬件提供的各种性能事件的支持；是CONFIG_BPF_EVENTS和CONFIG_UPROBE_EVENTS的依赖
10			CONFIG_HAVE_PERF_EVENTS	是CONFIG_PERF_EVENTS的依赖
11			CONFIG_PROFILING	启用分析器使用的扩展分析支持机制；是CONFIG_PERF_EVENTS的依赖
12	Cgroup		CONFIG_CGROUP_BPF	允许使用bpf(2) syscall命令BPF_PROG_ATTACH将eBPF程序附加到cgroup
13	Kprobes	4.1	CONFIG_KPROBE_EVENTS	允许用户通过ftrace接口动态添加跟踪事件，是CONFIG_BPF_EVENTS的依赖
14			CONFIG_KPROBES	允许在几乎任何内核地址捕获并执行回调函数
15			CONFIG_HAVE_KPROBES	CONFIG_KPROBES的依赖
16			CONFIG_HAVE_REGS_AND_STACK_ACCESS_API	是CONFIG_KPROBE_EVENTS的依赖
17	Uprobes	4.3	CONFIG_UPROBES	在用户空间启用检测应用程序
18			CONFIG_ARCH_SUPPORTS_UPROBES	是CONFIG_UPROBES以及CONFIG_UPROBE_EVENTS的依赖
19			CONFIG_UPROBE_EVENTS	允许用户通过跟踪事件接口在用户空间动态事件的基础上动态添加跟踪事件；是CONFIG_BPF_EVENTS的依赖
20			CONFIG_MMU	分页内存管理支持基于MMU的虚拟化寻址空间；是CONFIG_UPROBE_EVENTS的依赖
21	Tracepoints	4.7	CONFIG_TRACEPOINTS	支持跟踪点tracepoint
22			CONFIG_HAVE_SYSCALL_TRACEPOINTS	是CONFIG_TRACEPOINTS的依赖
23	bpf2bpf calls	4.16	/	/
24	Raw Tracepoints	4.17	同Tracepoints	/

使用eBPF的工业界项目

采集系统	简介	编程语言	eBPF库
Osquery (Facebook)	osquery is a SQL powered operating system instrumentation, **monitoring**, and analytics framework.	C++	ebpfpub 基于**libbpf**的C++库
Capsule8 (Capsule8 Inc)	**EDR**. Capsule8 performs advanced **behavioral monitoring** for cloud-native, containers, and traditional Linux-based servers.	Go	N/A
Sysmon for Linux (Microsoft)	Microsoft is working on an eBPF-based **monitoring** tool for Linux!	N/A	N/A
Tracee (Aqua Security)	Runtime **Security and Forensics** using eBPF	Go	libbpfgo 基于libbpf的Go库
Traceleft (ShiftLeft Inc)	eBPF based syscalls, files and network events tracing framework.	Go	Gobpf 基于BCC的Go库
FIM (DataDog Inc)	**File integrity monitoring** (FIM) is an invaluable tool for meeting regulatory frameworks that require routine file access auditing.	Go	N/A
Sysdig	Sysdig supports eBPF as an alternative to kernel instrumentation module.	N/A	N/A