TECHNICAL REFERENCE

AI in Cybersecurity

A Practical Reference for Offensive & Defensive Operations

Covering: LLM agents · MCP integration · Local inference · SLM farms · Binary analysis
Fuzzing · RAG & fine-tuning · ML-based defenses

Version 1.0 — May 2026

Offensive AI | Binary&Fuzzing | Local Inference | SLM Farm | ML Defense

Contents

1 Executive Summary **3**

2 AI-Augmented Offensive Security Tools **3**
2.1 LLM-Driven Payload Generation . . . 3
2.2 Agentic Pentest Frameworks . . . 3
2.3 AI for Exploit Development . . . 4

3 AI in Low-Level / Binary Security **4**
3.1 LLM Plugins for Disassemblers . . . 4
3.2 ML for Binary Vulnerability Discovery . . . 5
3.3 AI-Augmented Fuzzing . . . 5

4 Context-Aware Tool Use by AI Agents **5**
4.1 Tool Routing and ReAct . . . 5
4.2 Security-Specific Routing . . . 5
4.3 Guardrails . . . 6

5 Local Inference for Cybersecurity **6**
5.1 Runtimes . . . 6
5.2 Models for Security Work (mid-2026) . . . 6
5.3 Why Local Matters . . . 6

6 SLM Farm Architecture for Cybersecurity **7**
6.1 Concept and Motivation . . . 7
6.2 Specialist SLMs for Security . . . 7
6.3 Orchestrator Design . . . 7
6.4 SLM Farm Architecture Diagram . . . 9
6.5 Integration with the Pentest Pipeline . . . 9

7 MCP (Model Context Protocol) in Cybersecurity **10**
7.1 Architecture . . . 10
7.2 Security-Focused MCP Servers . . . 10
7.3 MCP Attack Surface . . . 10

8 Claude Code + Ollama Integration **10**
8.1 Setup (Ollama 0.14+) . . . 10
8.2 Cloud vs. Local Trade-Off . . . 11

9 Fine-Tuning vs. RAG for Security AI **11**
9.1 Decision Matrix . . . 11
9.2 RAG Pipeline Reference . . . 12
9.3 Domain Fine-Tuned Security Models . . . 12

10 Machine Learning in Cybersecurity (Defensive) **13**
10.1 Intrusion Detection . . . 13
10.2 Malware Classification . . . 13
10.3 UEBA and Commercial Landscape . . . 13
10.4 Phishing Detection . . . 13

11 Architecture Diagrams **13**

11.1 AI-Augmented Pentest Pipeline (Full) . . . 14
11.2 Local Inference Stack (Ollama + RAG + Agent) . . . 15
11.3 MCP Integration for Cybersecurity . . . 15
11.4 Fine-Tuning vs. RAG Decision Flowchart . . . 16
11.5 ML-Based Intrusion Detection Pipeline . . . 17

12 Recommendations & Staged Adoption Roadmap 17

13 Caveats & Research Integrity Notes 18

14 Emerging Frontiers: AI & ML in Low-Level Security 18
14.1 Binary Analysis and Reverse Engineering . . . 18
14.1.1 Automated Type Recovery in Stripped Binaries . . . 18
14.1.2 Cross-Architecture Binary Lifting and Normalisation . . . 19
14.1.3 Decompiler Output Quality Improvement . . . 19
14.1.4 Control Flow Integrity Violation Detection . . . 19
14.2 Vulnerability Discovery in Binaries . . . 19
14.2.1 ML-Guided Taint Analysis . . . 19
14.2.2 Heap Layout Prediction for Exploitation . . . 19
14.2.3 Patch Diffing and 1-Day Exploit Generation . . . 20
14.3 Hardware Security and Side-Channel Analysis . . . 20
14.3.1 Power and EM Side-Channel Attack Automation . . . 20
14.3.2 Fault Injection Targeting . . . 20
14.3.3 Hardware Trojan Detection in Netlists . . . 20
14.3.4 Microarchitectural Attack Surface Mapping . . . 20
14.4 Operating System and Kernel Security . . . 20
14.4.1 Kernel Vulnerability Discovery via Static Analysis + ML . . . 21
14.4.2 Syscall Sequence Anomaly Detection . . . 21
14.4.3 AI-Assisted Kernel Exploit Development . . . 21
14.4.4 OS Boot Chain Integrity Analysis . . . 21
14.5 Memory Forensics and Runtime Analysis . . . 21
14.5.1 Automated Memory Dump Triage with LLMs . . . 21
14.5.2 Heap Spray and ROP Chain Detection in Memory . . . 22
14.5.3 Kernel Object Integrity Verification . . . 22
14.6 Firmware and Embedded Systems . . . 22
14.6.1 Automated Firmware Unpacking and Analysis . . . 22
14.6.2 Bootloader Vulnerability Analysis . . . 22
14.6.3 RTOS Vulnerability Patterns . . . 22
14.6.4 Cross-Binary TPM and Secure Enclave Interaction Analysis . . . 22
14.7 AI-Augmented Exploit Primitives and Automation . . . 23
14.7.1 Automated ROP Chain Compilation . . . 23
14.7.2 Shellcode Polymorphism and Encoder Generation . . . 23
14.7.3 Format String and Integer Overflow Auto-Exploitation . . . 23

15 Research Directions: Cybersecurity Reasoning Systems 23
15.1 Taxonomy of Open Research Problems . . . 24
15.2 Direction 1: Neurosymbolic Reasoning for Exploit Analysis . . . 24
15.3 Direction 2: Causal Attack Graph Reasoning . . . 24
15.4 Direction 3: Multi-Agent Red/Blue Adversarial Self-Play . . . 25
15.5 Direction 4: Continual Learning for Threat Detection . . . 25
15.6 Direction 5: Explainable AI for Security Decisions . . . 26

15.7 Direction 6: Adversarial Robustness of the Reasoning System Itself 26
15.8 Research Agenda Summary 27

References & Further Reading **27**

Executive Summary

The adoption of large language models (LLMs) in cybersecurity workflows has accelerated sharply from 2024 to 2026. LLM-driven offensive security has shifted from single-prompt scripts to *agent harnesses* — multi-tool orchestrators that let a model plan, call tools such as `nmap`, `sqlmap`, and Burp, parse output, and iterate. The biggest practical gains are in reconnaissance, payload generation, and reverse engineering; novel exploit development at scale remains a future goal.

On the defensive side, the dominant architecture pairs classical ML (CNN/LSTM malware classifiers, GNN binary similarity, isolation-forest IDS, transformer phishing detectors) with LLM-driven SOC copilots backed by RAG over CVE/NVD/MITRE ATT&CK. Local inference via Ollama, MCP servers, and on-premise RAG is now production-viable for air-gapped and regulated environments.

Key Takeaways

- PentestGPT (USENIX Security 2024) is the canonical reference for autonomous LLM pentesters.
- MCP has become the dominant integration layer; Kali Linux ships `mcp-kali-server` officially.
- Ollama 0.14 added native Anthropic API compatibility, making Claude Code + local models viable.
- Fine-tune for *syntax and style*; use RAG for *facts that change daily*.
- Tool poisoning (Invariant Labs 2025, CyberArk 2025) is the most underappreciated MCP risk.
- SLM Farms (Section 6) offer a practical path to on-premise parallelism: eight 7B–13B specialists running simultaneously in the same VRAM envelope as one 70B model.

AI-Augmented Offensive Security Tools

LLM-Driven Payload Generation

LLMs generate context-aware payloads by reading the target's request/response, framework fingerprints, and prior responses rather than iterating over a static wordlist. The 2023 paper *Prompt Injection attack against LLM-integrated Applications* (arXiv:2306.05499) models prompt injection as a *Framework–Separator–Disruptor* composition, achieving 86.1% success across 36 LLM-integrated applications. Snyk demonstrated the inverse: jailbreaking an LLM into emitting `'); DROP TABLE users; -` as an SQL injection payload generator.

Key Tools

- **PROMPTFUZZ** (arXiv:2409.14729) — AFL-style mutation/feedback for prompt-injection testing.
- **Nuclei AI** (ProjectDiscovery) — generates Nuclei YAML templates from CVE descriptions.
- **BurpGPT** — LLM-driven passive scanner for Burp Suite.
- **AutorizePro / ReconAIzer** — Burp extensions for authorization testing and recon.

Agentic Pentest Frameworks

The canonical architectures converge on three components: a *reasoner* that maintains the attack tree, a *generator* that emits exact commands, and a *parser* that consumes tool output.

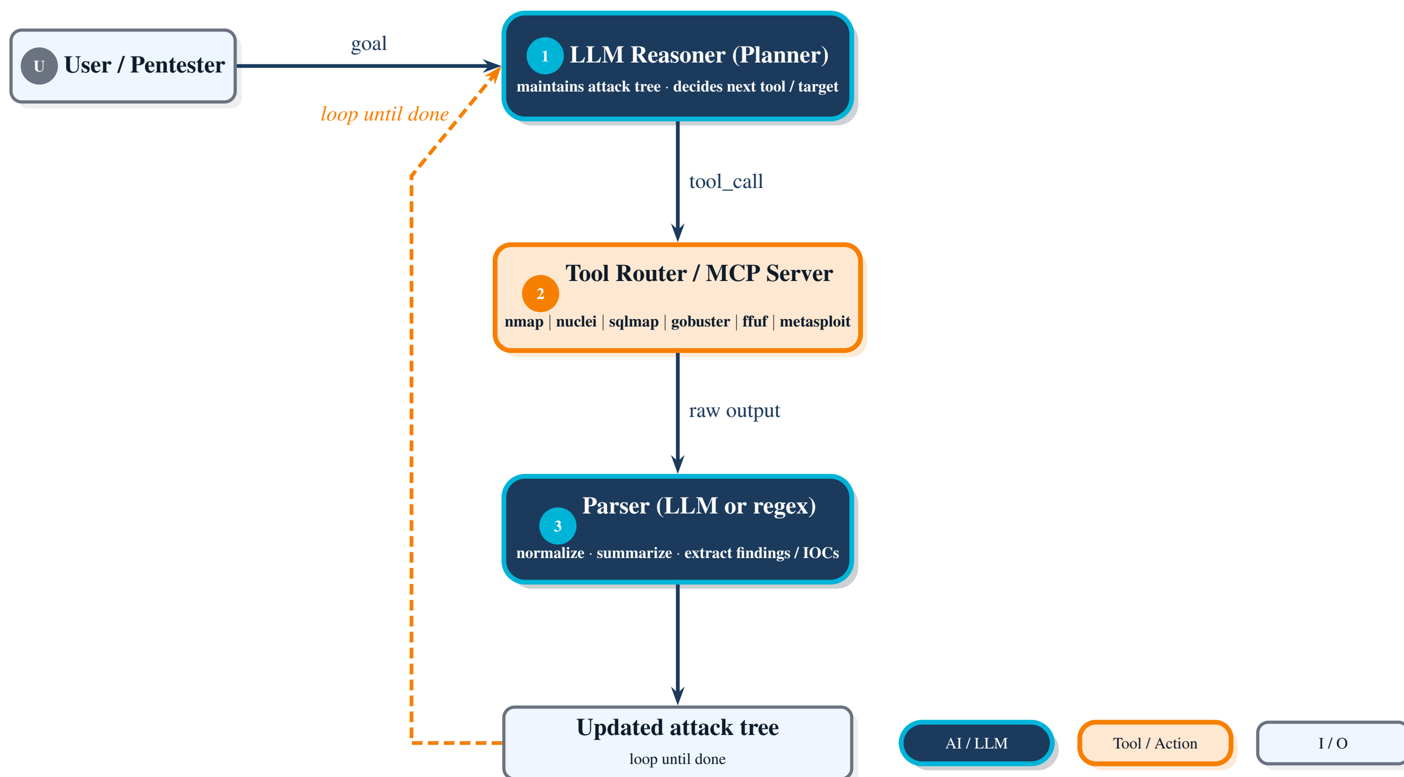

Figure 1. Core three-step agent loop for automated penetration testing. The reasoner plans, the tool router executes, and the parser normalises output before the next iteration. See Figure 4 for the expanded pipeline with guardrails.

Notable frameworks (2024–2026).

- **PentestGPT** — Three-module architecture (reasoning / generation / parsing), now Docker-first with 104 XBOW benchmark validations.
- **HackingBuddyGPT** — Linux privesc + web API pentesting; logs every step to SQLite; supports OpenAI or local Ollama backends.
- **HexStrike-AI** (`0x4m4/hexstrike-ai`) — Multi-agent MCP server exposing 150+ tools: BugBounty Agent, CTF Solver, CVE Intelligence, Exploit Generator.
- **vulnhuntr (Protect AI)** — Zero-shot vulnerability discovery in Python/C#/Go codebases; found 0-days in gpt_academic, FastChat, and RAGFlow.
- **BurpGPT, AutorizePro, ReconAIzer** — Burp Suite extensions leveraging LLMs.

AI for Exploit Development

LLMs remain poor at novel exploit primitives but are useful for: shellcode adaptation and encoder selection; ROP gadget chaining over a known gadget set; translating PoC code between languages; and writing Metasploit modules from CVE descriptions. Production teams treat the LLM as a junior collaborator that drafts while a human validates.

AI in Low-Level / Binary Security

LLM Plugins for Disassemblers

A mature ecosystem of LLM plugins now integrates into IDA Pro and Ghidra, most supporting fully local operation via Ollama or LM Studio.

Tool	Host	Capabilities	Backend
Gepetto	IDA Pro ≥7.6	Explain function, rename variables	OpenAI / Anthropic / local
IDA Pro MCP	IDA Pro	Bridges IDA to any MCP client	Any MCP client
IDAssist	IDA Pro 9.0+	Dockable panel, knowledge graph, RLHF export	Ollama / LiteLLM
GhidrAssist + MCP	Ghidra	ReAct orchestrator, RAG + MCP tabs	Ollama / LM Studio / Claude
GhidraMCP (LaurieWired)	Ghidra	Most-cited MCP-for-RE bridge	Claude Desktop
OGhidra (LLNL)	Ghidra	Conversational + Investigation modes, blackboard	Ollama
GhidrOllama	Ghidra	Lightweight; supports remote Ollama	Ollama (any GGUF)
Sidekick	Binary Ninja	Function summaries, search (commercial)	Cloud LLM

Table 1. LLM plugins for disassemblers (mid-2026).

ML for Binary Vulnerability Discovery

- **CodeArt** (FSE 2024) — BERT-like model pre-trained on 26M stripped binary functions with attention regularization from program-dependence transitive closures. Reports gains of 53→64% binary similarity, 74→94% malware family classification.
- **VulHawk** (NDSS 2023) — Lifts binaries to Hex-Rays microcode, uses RoBERTa-variant embeddings, then GCNs over CFGs; outperforms Asm2Vec, BinDiff, SAFE, Trex across 72 file environments.
- **BinaryAI** (ICSE 2024, Tencent Keen Lab) — Two-phase binary-to-source matching: transformer embeddings retrieve top-K source functions, then locality-driven matching. Used for binary SCA/TPL detection.

AI-Augmented Fuzzing

Contested Claims

NEUZZ (IEEE S&P 2019) reported 3× more edge coverage than gray-box fuzzers at 24 h and found 31 unknown bugs. However, the Neuzz++ follow-up (ESEC/FSE 2023, Bosch Research) found “the original performance claims for NPS fuzzers do not hold. . . standard gray-box fuzzers almost always surpass NPS-based fuzzers.” **There is no built-in neural mode in AFL++ today** — only a generic custom-mutator API. **CodaMOSA** (ICSE 2023) is the most credible current AI/fuzzing win.

Context-Aware Tool Use by AI Agents

Tool Routing and ReAct

The dominant pattern is **ReAct** (Reason → Act → Observe) loops. The LLM is given a tool catalog (JSON schemas), a system prompt with policy constraints, and a scratchpad of prior observations. The agent terminates when: no tool call is emitted; maximum turns are exceeded; a guardrail trips; or the model issues a safety refusal.

Security-Specific Routing

A well-built pentest agent uses *phase-gated* tool routing:

Phase	Allowed Tools
Recon	nmap, masscan, subfinder, amass, theharvester, shodan-mcp
Enumeration	gobuster, ffuf, feroxbuster, nikto, wpscan
Vuln Scanning	nuclei, sqlmap, dalfox, wafw00f
Exploitation	metasploit, searchsploit, hydra, crackmapexec
Post-exploit	evil-winrm, john, hashcat, mimikatz wrappers
Reporting	Markdown writer, Jira, DefectDojo

Guardrails

A standard pre/post execution hook wraps every tool call:

Listing 1. Pre- and post-execution guardrail hooks.

```
@before_tool
def scan_inputs(call):
    if matches_prompt_injection(call.args): raise GuardrailTrip()
    if call.tool not in allowlist[phase]:   raise PolicyDeny()
    if target_out_of_scope(call.args):      raise ScopeViolation()

@after_tool
def scan_outputs(result):
    redact_pii(result)
    strip_hidden_instructions(result)
    if exfil_pattern(result): alert_and_quarantine()
    return result
```

Local Inference for Cybersecurity

Runtimes

Ollama (v0.14+ with native Anthropic-compatible API) is the most widely used local runtime for security workflows, followed by **llama.cpp** (CPU/Metal/CUDA/ROCm, GGUF quantized), **LM Studio** (desktop GUI), and **vLLM/TGI/SGLang** for higher-throughput on-premise servers.

Models for Security Work (mid-2026)

Model	Best for	Notes	VRAM (Q4)
WhiteRabbitNeo / Deep Hat	Offensive/defensive cyber	Uncensored, cyber-tuned	7B ≈5 GB
DeepSeek-Coder V2 16B	Exploit scripting, IaC review	Strong function calling	≈10 GB
Qwen2.5-Coder 32B	RE code understanding	Best code quality at size	≈20 GB
Llama 3.3 70B	General SOC analyst tasks	Closest to cloud quality	≈40 GB
Mistral 7B	Log / IOC extraction	Lightweight, fast	≈5 GB
GLM-4.7 Flash	Agentic coding + tool calls	20K context, MoE	≈15 GB

Table 2. Local models suited to cybersecurity work.

Why Local Matters

GDPR Article 35 DPIA risk, Article 83(5) penalties (up to €20M or 4% global turnover), data-residency laws, ITAR/CUI controls, and the basic fact that sensitive binaries, exploit PoCs, and incident logs should not be uploaded to a third party are the primary drivers. Local inference sidesteps cross-border transfer entirely.

SLM Farm Architecture for Cybersecurity

Concept and Motivation

Rather than routing every security task through one large model, the **SLM Farm** pattern deploys a cluster of independently fine-tuned Small Language Models (SLMs, typically 3B–13B parameters each) running in parallel on the same hardware pool. Each SLM is trained for one narrow job and does it extremely well. An orchestrator dispatches incoming work to the correct specialist, collects results, and synthesises a unified output.

The pattern is compelling for cybersecurity specifically because security work decomposes naturally into dozens of tight sub-tasks (log parsing, CVE triage, payload mutation, binary function naming, YARA rule authoring) that share almost no vocabulary or reasoning style with one another. A generalist 70B model is wasteful for "extract IOCs from this Zeek log"; a 7B model fine-tuned on 500 k labelled SIEM lines will be faster, cheaper, and more accurate.

Why SLM Farms Beat a Single Large Model for Security

Specialisation beats generalisation for narrow tasks — a 7B model fine-tuned on malware reports outperforms a 70B generalist on malware classification. **Parallelism** allows log parsing, CVE lookup, and initial report drafting to run simultaneously. **Cost efficiency** means eight 7B models at Q4_K_M consume ≈40 GB VRAM total — same as one 70B model — but can process eight tasks concurrently. **Failure isolation** ensures one crashing SLM does not abort the whole pipeline. **Auditability** is improved because each SLM's input and output is a discrete, logged, reviewable step with no cross-task context contamination.

Specialist SLMs for Security

Each SLM in the farm should be fine-tuned on a domain-specific corpus using LoRA or QLoRA (4-bit base, 16-bit adapters) so training fits on a single consumer GPU. The table below lists recommended specialisations for a production security farm.

SLM Name	Base / Size	Training corpus	VRAM (Q4)
SLM-Log	Mistral 7B	SIEM logs, Zeek/Suricata alerts, syslog	≈5 GB
SLM-CVE	Mistral 7B + RAG	NVD JSON, EPSS, CISA KEV, CVSS advisories	≈5 GB
SLM-Payload	WRN 13B	Public PoCs, pentest payloads, CTF write-ups	≈9 GB
SLM-Decompile	CodeLlama 7B	Binary–source function pairs (BinaryAI corpus)	≈5 GB
SLM-YARA	Qwen2.5-Coder 7B	YARA rules, Sigma rules, IOC-to-rule examples	≈5 GB
SLM-Report	Llama 3.2 3B	Pentest reports, CVE advisories, vuln write-ups	≈2 GB
SLM-Phish	DistilBERT 0.3B	PhishTank, OpenPhish, ISCX-URL corpora	<1 GB
SLM-Triage	Mistral 7B	CVSS scores, EPSS, KEV, alert severity labels	≈5 GB

Table 3. Recommended SLM specialisations for a security farm. Total VRAM for all eight: ≈37 GB — fits in a dual RTX 3090 or a single A100.

Orchestrator Design

The orchestrator can itself be lightweight — a deterministic rules engine, a small classifier, or a thin LLM (e.g. Phi-3 Mini 3.8B) that reads the task description and emits a routing token. Crucially, the orchestrator does *not* need to understand the domain of each specialist; it only needs to recognise which *category* of task is being requested. This makes the routing layer extremely fast and auditable.

For tasks that require multiple specialists simultaneously (e.g. "analyse this suspicious binary and draft a finding") the orchestrator fans out to SLM-Decompile and SLM-Report in parallel, then the aggregator merges their outputs before sending to the analyst. The aggregator is typically a template engine or a thin summariser SLM.

Listing 2. Minimal SLM-farm dispatcher in Python.

```
ROUTING_TABLE = {
    "log_analysis":      "slm-log:7b",
    "cve_triage":        "slm-cve:7b",
    "payload_gen":       "slm-payload:13b",
    "binary_naming":     "slm-decompile:7b",
    "yara_generation":   "slm-yara:7b",
    "report_writing":    "slm-report:3b",
    "phishing_check":    "slm-phish:0.3b",
    "alert_triage":      "slm-triage:7b",
}

async def dispatch(task_type: str, payload: str) -> str:
    model = ROUTING_TABLE[task_type]
    # All models expose the same OpenAI-compat /v1/chat endpoint
    response = await ollama_client.chat(model=model,
                                        messages=[{"role": "user",
                                                   "content": payload}])
    return response["message"]["content"]

async def run_farm(tasks: list[dict]) -> list[str]:
    # Fan out all tasks in parallel
    return await asyncio.gather(*[dispatch(t["type"], t["data"])
                                  for t in tasks])
```

SLM Farm Architecture Diagram

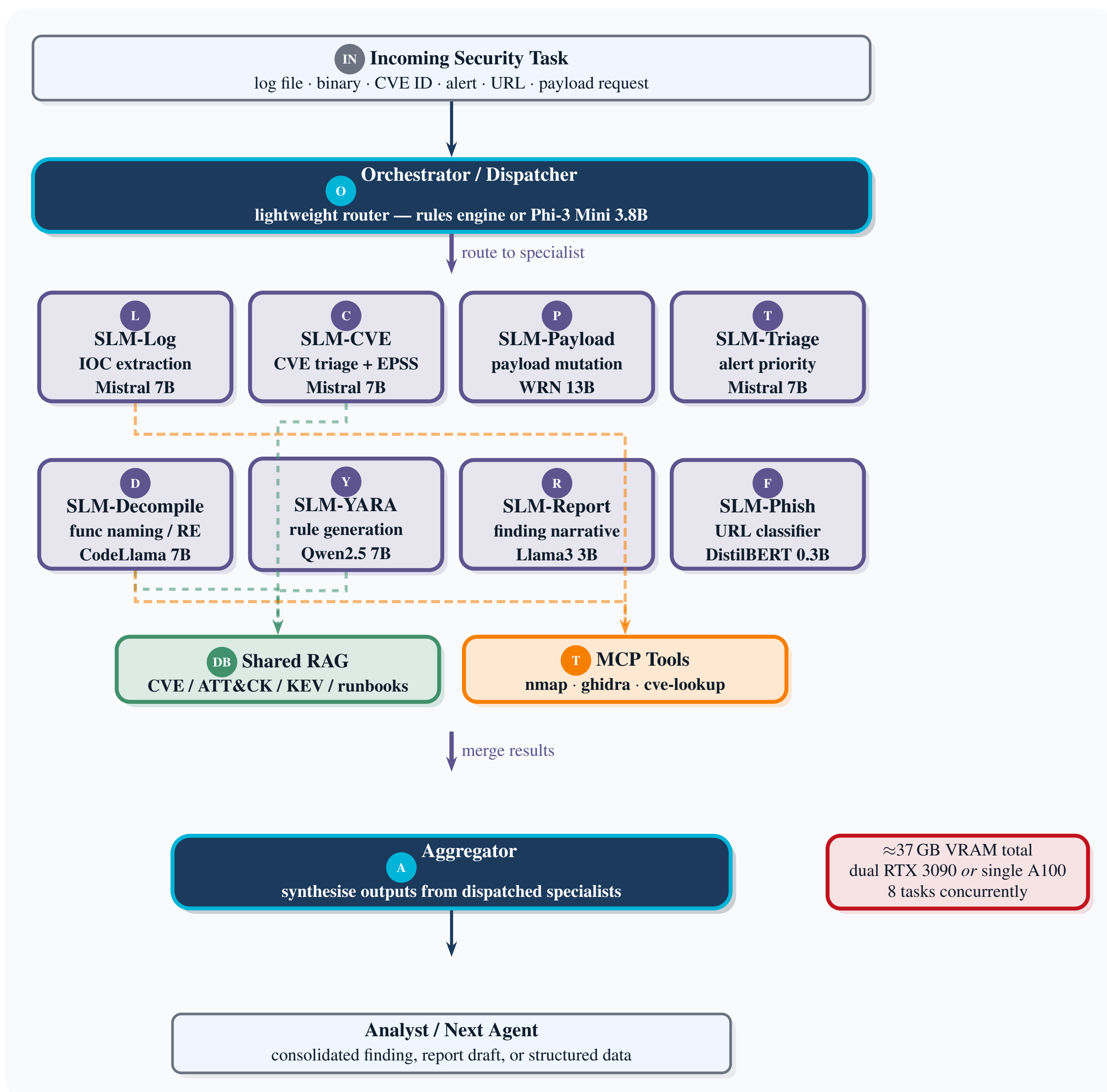

Figure 2. SLM Farm architecture: eight specialist models run in parallel inside the farm boundary. A single dispatch arrow from the orchestrator routes the task; a single result arrow carries the merged output to the aggregator. Dashed lines show optional shared-infra usage (RAG / MCP tools) without crossing the dispatch paths.

Integration with the Pentest Pipeline

The SLM farm sits naturally between the MCP tool execution layer and the human analyst in the full pentest pipeline. When nmap returns a scan, SLM-Log extracts open ports and service fingerprints while SLM-CVE simultaneously looks up known CVEs for each detected service version. When a binary is submitted for reverse engineering, SLM-Decompile names functions while SLM-Report begins drafting the finding skeleton. Both complete within seconds; the aggregator assembles a unified brief in under ten

seconds on consumer hardware.

SLM Farm vs. Single Large Model: Practical Guidance

Use an SLM farm when: (a) you have well-defined, repetitive sub-tasks; (b) privacy requires all processing to stay on-premise; (c) you need parallel processing of many inputs simultaneously; or (d) you have labelled training data for specific security domains. Prefer a single large model (cloud or 70B local) when: (a) you need novel cross-domain reasoning (e.g. developing a new attack chain); (b) the task requires long-horizon planning; or (c) you do not yet have enough labelled data to fine-tune a specialist. The best production setups use *both*: the SLM farm handles high-volume, well-defined tasks at speed, and a 70B generalist or cloud LLM handles the hard, novel, or cross-domain cases that fall through.

MCP (Model Context Protocol) in Cybersecurity

Architecture

MCP is a JSON-RPC 2.0 protocol over stdio or HTTP/SSE. Standard methods include `tools/list`, `tools/call`, `resources/read`, and `prompts/list`. Tool descriptions are injected into the LLM's context window and thus function as *executable influence*, not mere metadata.

Security-Focused MCP Servers

- **mcp-kali-server** — Official Kali package; Flask bridge on port 5000; runs nmap, Nikto, sqlmap, wpscan, hashcat, john on demand.
- **HexStrike-AI MCP server** — 150+ tools across network, web, password, and binary RE categories.
- **cve-mcp-server** (`mukul975`) — 27 tools across 21 APIs: NVD lookup, EPSS scoring, CISA KEV, GitHub PoC discovery, Shodan, VirusTotal.
- **GhidraMCP / IDA Pro MCP / GhidrAssistMCP** — RE-specific MCP servers.

MCP Attack Surface

Security Warning: MCP Risks

- **Tool poisoning (Invariant Labs 2025):** Malicious instructions hidden in a tool's `description` field — invisible to humans, visible to the LLM. Bypasses all input validation.
- **Rug pull:** Server serves a clean description at install, swaps to malicious later.
- **Confused deputy:** Over-permissioned tools acting on behalf of the user.
- **Indirect prompt injection via outputs (CyberArk "Poison everywhere"):** Any auto-generated field can carry hidden instructions.

Mitigations: OWASP MCP Security Cheat Sheet; deny-by-default tool invocation; per-tool allowlists; MCP-Scan (Invariant); digital signing and version-locking of tool descriptions.

Claude Code + Ollama Integration

Setup (Ollama 0.14+)

Ollama 0.14 added native Anthropic Messages API compatibility, so Claude Code's agentic loop can be pointed at it directly:

Listing 3. Point Claude Code at a local Ollama instance.

```
ollama pull qwen3-coder   # or glm-4.7-flash, deepseek-coder-v2
export ANTHROPIC_AUTH_TOKEN=ollama
export ANTHROPIC_API_KEY=""
```

```
export ANTHROPIC_BASE_URL=http://localhost:11434
claude --model qwen3-coder
```

For multiple backends, route through **LiteLLM** as an Anthropic-format proxy (see config in Section 11.2).

Cloud vs. Local Trade-Off

The `paddo.dev` head-to-head benchmark: a .NET monorepo entity-trace task ran in **1m 13s on cloud Claude vs. 1h 22m on local GLM-4.7 Flash** — a 68× latency penalty with "nearly identical" output quality. The pragmatic posture: cloud Claude for novel reasoning and tight loops; local Ollama for sensitive code, offline operations, and background bulk analysis.

Fine-Tuning vs. RAG for Security AI

Decision Matrix

Need	Why	Approach
Custom exploit/payload syntax	Style and vocabulary live in weights	Fine-tune (LoRA)
Company-specific vuln patterns	Patterns in weights; instances in vector store	Fine-tune + RAG
Malware family classification	Discriminative task, fixed label space	Fine-tune classifier head
CVE / NVD / KEV / EPSS lookup	Updated daily — never train on it	RAG
MITRE ATT&CK technique mapping	Hierarchical, evolving, citation-required	RAG
Internal CMDB / SIEM context	Live data, access-controlled	RAG (via MCP)
Hardening guides (NIST 800-53)	Auditable, version-pinned	RAG
Triage prose generation	Tone and report structure	Fine-tune (small)

Table 4. Decision matrix for fine-tuning vs. RAG in security AI.

RAG Pipeline Reference

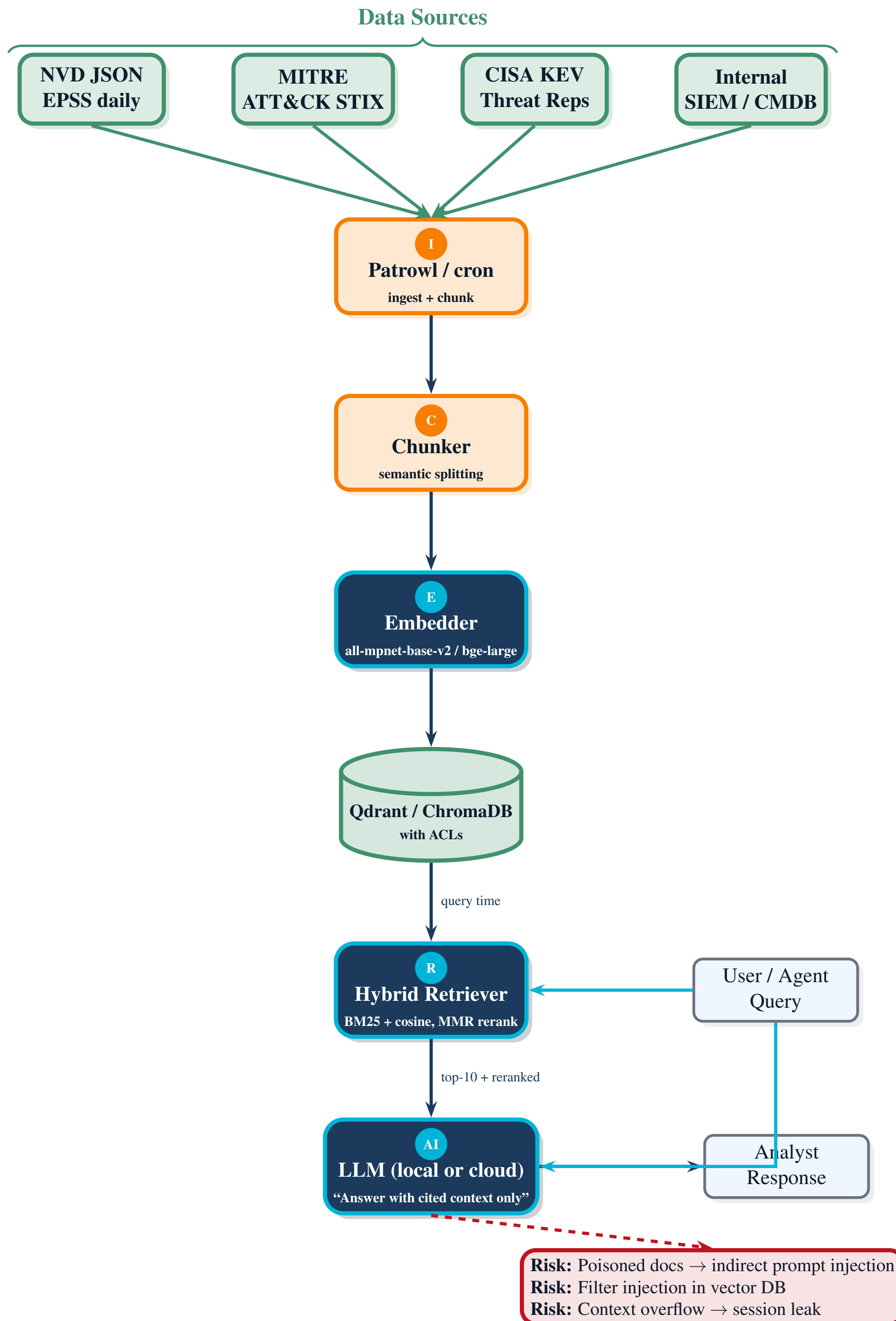

Figure 3. RAG pipeline for cybersecurity intelligence (CVE/NVD/ATT&CK/KEV/internal).

Domain Fine-Tuned Security Models

- **SecureBERT 2.0** (arXiv:2510.00240) — ModernBERT-based, hybrid NL+code tokenization; SOTA on cyber semantic search, NER, and vulnerability detection.

- **WhiteRabbitNeo / Deep Hat** (Kindo) — 7B/13B offensive-cyber tuned; available on Ollama.
- **Sec-Gemini v1** (Google, April 2025) — Integrates GTI, OSV, and Mandiant Threat Intelligence as near-real-time RAG; reports 11%+ gain on CTI-MCQ benchmark.
- **CyberSecEval 2** (Meta) — Evaluation benchmarks for security-tuned LLMs.

Machine Learning in Cybersecurity (Defensive)

Intrusion Detection

Anomaly detection uses isolation forests, one-class SVMs, and autoencoders on NetFlow/Zeek logs. Supervised classifiers (XGBoost, LightGBM) are trained on CIC-IDS-2017 / UNSW-NB15 / IoT-23 features. Graph-based GNNs over communication graphs are emerging for lateral-movement and APT detection. Concept drift is handled through periodic retraining on rolling windows.

Malware Classification

- **CNN on binary visualizations:** Convert PE bytes to 2D grayscale images; classify with ResNet/Swin-T. CNN-BiLSTM hybrid (MDPI 2024) reports >99% on Microsoft Malware Challenge.
- **BiLSTM on API call sequences:** Cuckoo Sandbox extracts behavioral sequences; BiLSTM captures forward+backward dependencies.
- **Hybrid CNN-LSTM-BERT** (ScienceDirect 2024): opcode 8-grams + dynamic execution; ≈99.4% on polymorphic malware.

UEBA and Commercial Landscape

Vendor	Strength	Approach
CrowdStrike Falcon	Endpoint, XDR, agentic SOC	Threat Graph: 1T+ events/day, 2T vertices, 15 PB
Darktrace	NDR, email, OT	Self-learning unsupervised baselines per organization
Vectra AI	NDR, identity, cloud	35 patented attacker-behavior models
SentinelOne Singularity	Endpoint + autonomous response	Static + behavioral AI per agent
Microsoft Sentinel + Defender	SIEM/XDR + Security Copilot	ML anomalies + LLM SOC copilot
Open-source (Wazuh / Suricata)	DIY / SIEM	Zeek, Elastic, OpenSearch, Splunk MLTK

Table 5. Commercial AI-driven security platforms (mid-2026).

Phishing Detection

DistilBERT/BERT URL classifiers outperform classical SVM/RF. Hybrid models combine email body (BERT sentence embeddings) with structural URL features (length, IP-vs-domain, special chars). Production stacks (Microsoft Defender, Proofpoint, Abnormal Security) layer transformer text classifiers with image-based brand-impersonation detection and sender-reputation graphs.

Architecture Diagrams

AI-Augmented Pentest Pipeline (Full)

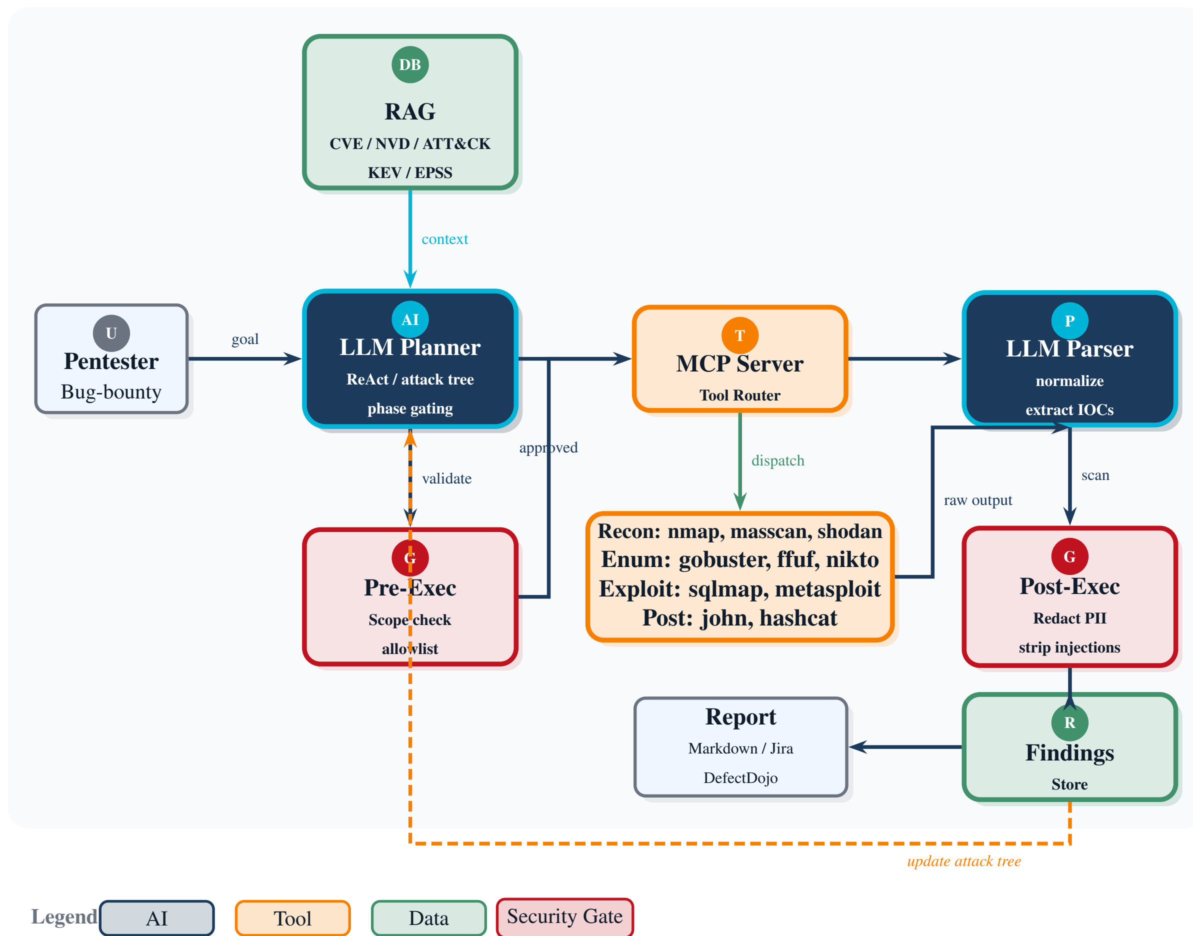

Figure 4. Complete AI-augmented penetration testing pipeline with guardrails and RAG.

Local Inference Stack (Ollama + RAG + Agent)

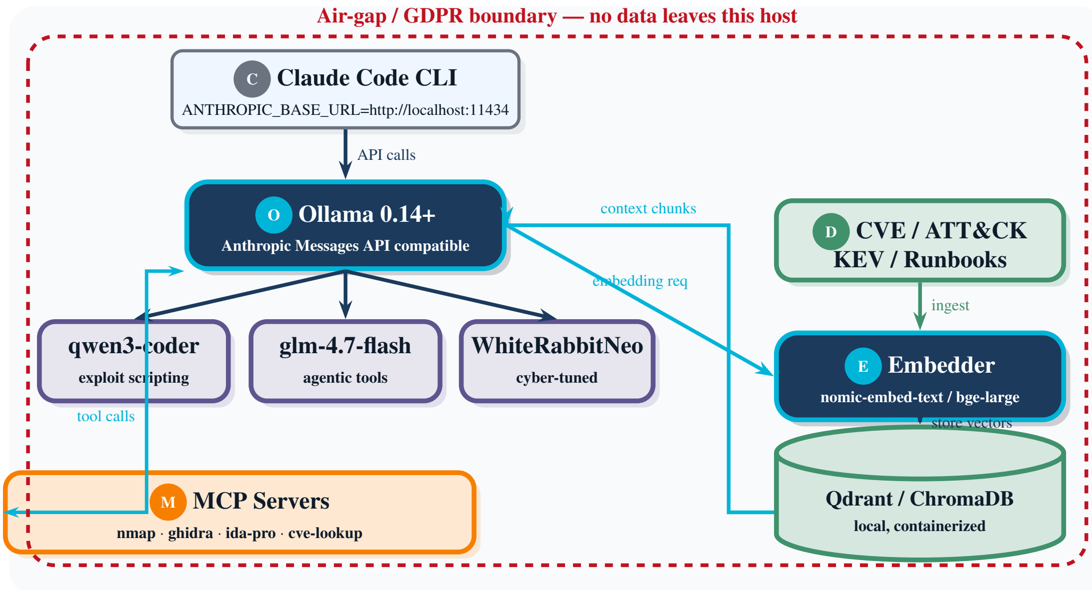

Figure 5. Local inference stack: Claude Code + Ollama + on-premise RAG + MCP tools. Left column handles inference; right column handles retrieval. Both columns run entirely on the same host behind the GDPR boundary.

MCP Integration for Cybersecurity

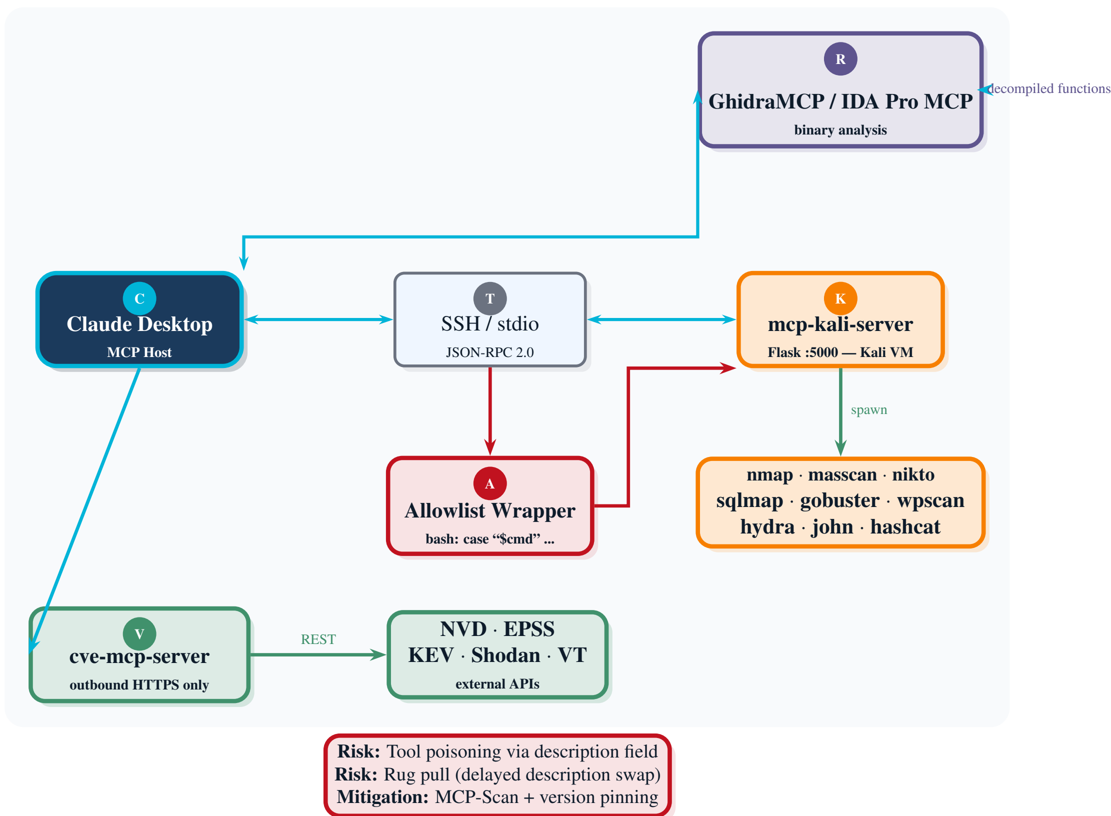

Figure 6. MCP integration topology for cybersecurity: Claude Desktop + Kali tools + CVE APIs + RE.

Fine-Tuning vs. RAG Decision Flowchart

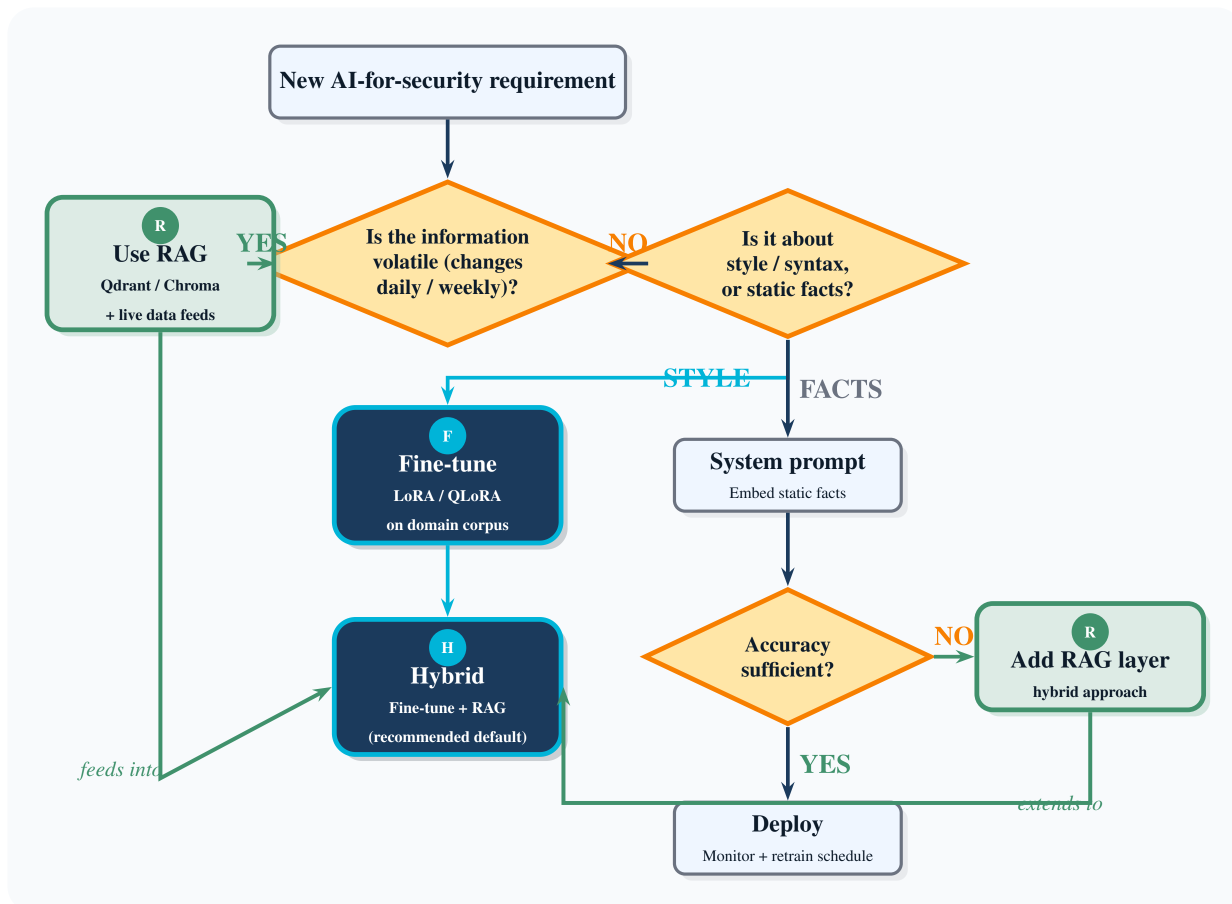

Figure 7. Decision flowchart for choosing between fine-tuning, RAG, and hybrid approaches. All connections are routed as clean L-shapes: no diagonal lines.

ML-Based Intrusion Detection Pipeline

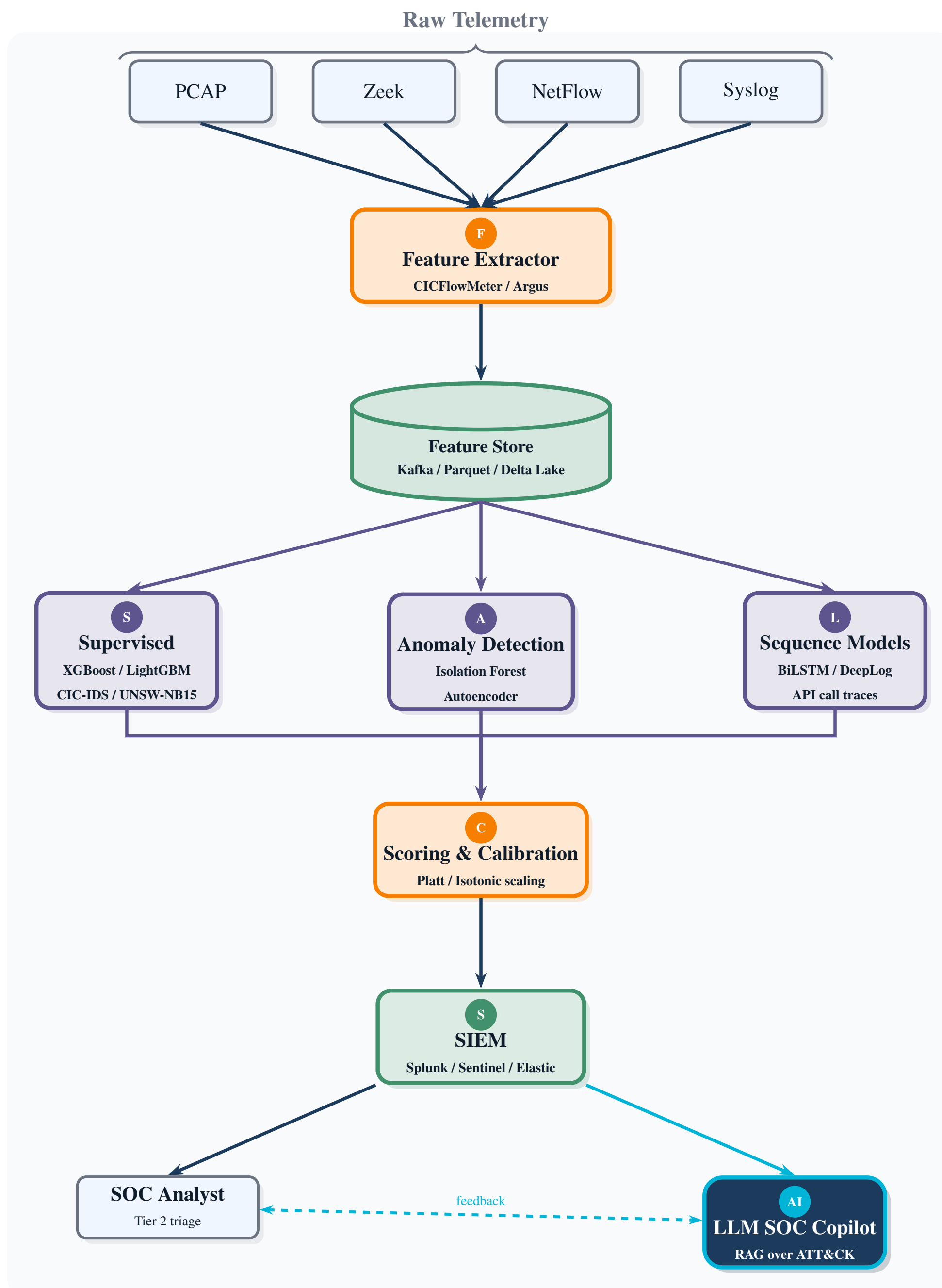

Figure 8. ML-based network intrusion detection pipeline from raw telemetry to SOC.

Recommendations & Staged Adoption Roadmap

Stage 1 — Pilot (weeks 1–4). Stand up Ollama 0.14 + Claude Code on a single workstation. Pull `qwen3-coder`, `deepseek-coder-v2`, and `whiterabbitneo` (13B Q4_K_M). Install GhidrAssist + GhidraMCP and run a known CTF binary end-to-end. Spin up a CVE RAG with Qdrant + NVD JSON + EPSS + MITRE ATT&CK STIX. Adopt OWASP MCP Security Cheat Sheet controls before exposing any MCP server beyond localhost.

Stage gate: >70% useful-output rate on a 20-binary RE corpus, judged by a senior reverser.

Stage 2 — Build (months 2–4). Deploy PentestGPT or HexStrike-AI in a network-isolated Kali VM with the documented MCP allowlist wrapper. Stand up vulnhuntr on a CI mirror of your Python/Go/Java services (confidence $\geq 8 \Rightarrow$ P2 ticket). Wire `cve-mcp-server` into your Tier-2 SOC analyst workflow; measure mean-time-to-triage vs. baseline.

Stage gate: $\geq$30% reduction in MTTR for CVE triage; zero MCP tool-poisoning incidents.

Stage 3 — Operationalize (months 4–9). Fine-tune (LoRA, $\sim$1–3M tokens) a 7B–13B base on internal incident write-ups for report-tone generation; keep all factual claims in RAG. Replace generic anomaly detection with a DistilBERT URL classifier + BiLSTM API-sequence classifier and benchmark against commercial vendors *for your environment specifically* before renewing contracts.

Caveats & Research Integrity Notes

Important Caveats

- **Vendor statistics are not independent research.** Vectra's "85%+ alert reduction" and Aikido's "9 out of 10 practitioners believe AI will take over pentesting" come from self-commissioned reports. Treat any percentage from a vendor's own blog as advertising.
- **Neural fuzzing claims are contested.** NEUZZ's 2019 S&P results were not reproduced by Bosch Research in ESEC/FSE 2023. Standard gray-box fuzzers often outperform neural-program-smoothing fuzzers.
- **vulnhuntr, PentestGPT, and HexStrike-AI emit confident output that is frequently wrong.** Confidence $\geq$8 means *worth investigating*, not *confirmed bug*.
- **MCP is young and rough.** OWASP, Invariant Labs, CyberArk, and Wallarm have all published serious security risks against MCP in 2025 alone. Never connect an agent to an unvetted MCP server.
- **Local LLM agent loops break silently** when models do not emit tool-call JSON in the exact expected format. Multiple Ollama community reports confirm Claude Code on local Qwen/DeepSeek can degrade to plain-chatbot mode without visible error.
- **CodeArt and VulHawk results are author-reported.** Independent replication of binary-similarity SOTA gains is uneven.
- **Sec-Gemini's fine-tuning architecture is not fully disclosed.** Google's April 2025 blog confirms RAG over GTI/OSV/Mandiant but omits fine-tuning specifics.

Emerging Frontiers: AI & ML in Low-Level Security

The deepest and most under-explored applications of AI in cybersecurity are at the *lowest levels of the stack*: raw binaries, processor microarchitecture, operating-system internals, memory forensics, bootloaders, and hardware itself. These domains resist classical automation because they lack human-readable structure, have enormous state spaces, and require reasoning across multiple abstraction layers simultaneously. AI and ML are uniquely suited to bridge these gaps.

Binary Analysis and Reverse Engineering

Beyond the LLM plugins for IDA Pro and Ghidra already covered in §2, the following deeper problems remain open:

Automated Type Recovery in Stripped Binaries

Stripped binaries lose all type information at compile time. Recovering it is essential for understanding data structures, especially in malware and closed-source libraries. Current SOTA (DIRTY, EKLAVYA, TypeScript) trains neural networks on debug-symbol pairs, but accuracy drops sharply for complex pointer-

to-struct chains, unions, and recursive types. Open problems: generalising across compilers (`gcc`, `clang`, MSVC) and optimisation levels (`-O0` through `-O3`), and handling LLVM bitcode obfuscation (OLLVM, Tigress).

Cross-Architecture Binary Lifting and Normalisation

A vulnerability found in an ARM Cortex-M firmware may also exist in the same library compiled for x86_64 or RISC-V. **Binary lifting** — translating machine code to an intermediate representation (LLVM IR, VEX, ESIL) — enables cross-architecture comparison. ML applied here: learning architecture-agnostic embeddings of basic blocks (similar to `asm2vec` and `Trex`) that map semantically equivalent code to nearby points in embedding space regardless of ISA. The practical goal is one model that can match a CVE-affected function across all architectures in a firmware corpus without recompiling.

Decompiler Output Quality Improvement

Hex-Rays and Ghidra's decompiler produce pseudocode that is structurally correct but semantically opaque: variables named `v4`, `v12`, `puVar3`; lost loop semantics; incorrect type casts. LLMs fine-tuned on (assembly, clean C source) pairs can recover: meaningful variable and parameter names; data-structure semantics ("this is a linked-list node, not a raw 8-byte struct"); algorithmic intent ("this is an RC4 key-scheduling loop"). LLM4Decompile (arXiv:2403.05286) is the most credible 2024 result: a 33B CodeLlama variant fine-tuned on 4 billion tokens of assembly–source pairs, achieving 21.7% re-compilability on Decompile-Eval. Remaining gap: handling compiler-synthesised code (vtable dispatch, RTTI, exception unwinding tables) that has no direct source-level analogue.

Control Flow Integrity Violation Detection

CFI (Control Flow Integrity) defences (clang CFI, Microsoft CFG, grsecurity RAP) restrict indirect branches to a set of valid targets. Attackers bypass CFI by reusing *valid* targets in unexpected sequences (COOP — Counterfeit Object-Oriented Programming). ML can model the *expected distribution of call sequences* at runtime and flag deviations even when each individual transfer is CFI-compliant. LSTM/Transformer models trained on normal execution traces of a target application can detect COOP-style attacks with low false-positive rates on synthetic benchmarks; generalisation to production workloads is still an open problem.

Vulnerability Discovery in Binaries

ML-Guided Taint Analysis

Classical taint analysis (tracking attacker-controlled data from source to sink) is precise but computationally expensive and produces many false positives from over-approximation. ML can play two roles: (1) a *pre-filter* that ranks functions by likelihood of containing a taint flow, so that precise analysis is only run on the top-k candidates; (2) a *false-positive reducer* that reads the full taint trace and the surrounding code context and classifies it as exploitable or not. Fine-tuning CodeBERT on (taint trace, exploitability label) pairs is a tractable research project with high practical value.

Heap Layout Prediction for Exploitation

Many modern exploits (use-after-free, heap overflow) depend on reliably shaping the heap so that a victim allocation lands adjacent to a controlled one. Today this requires manual trial-and-error or application-specific allocator knowledge. ML trained on heap allocation traces can learn *allocator behaviour models*: given the sequence of `malloc`/`free` calls so far, predict the address of the next allocation with useful probability. This collapses the manual heap-grooming phase of exploit development into an automated step.

Patch Diffing and 1-Day Exploit Generation

Between a vendor releasing a security patch and the last unpatched system, there is a window where an attacker who can read the patch can generate an exploit faster than defenders can deploy the fix. ML can accelerate this: given two binary versions of a library (pre/post patch), a model can identify the changed function, infer the vulnerability class from the diff pattern, and generate an initial PoC. vulnhuntr already does this at source level; binary-level equivalents (BinDiff + LLM reasoning over the diffed pseudocode) are an active research area.

Hardware Security and Side-Channel Analysis

Power and EM Side-Channel Attack Automation

Side-channel attacks (SCA) exploit physical information leakage — power consumption, electromagnetic radiation, timing — to recover secret keys from cryptographic implementations. Traditional SCA (CPA, DPA, DEMA) requires expert knowledge of the leakage model. **Deep learning SCA** (DLSCA) trains CNNs or MLPs directly on raw power traces, eliminating the need for a leakage model. Results on AES implementations (ASCAD dataset) show that a CNN with three convolutional layers recovers the key in under 1,000 traces on unprotected targets. Open problems: attacking higher-order masked implementations; generalising across different chips of the same family; and automating the measurement setup (trace alignment, trigger selection) with ML rather than manual configuration.

Fault Injection Targeting

Fault injection attacks (voltage glitching, clock glitching, laser fault injection) disturb execution to skip instructions or corrupt registers — a technique used to bypass secure-boot checks, extract keys from TPM chips, and defeat anti-tamper logic. Today, finding the correct fault parameters (voltage level, duration, timing offset) requires hundreds of hours of manual tuning. Bayesian optimisation and reinforcement learning can automate this search: the RL agent selects fault parameters, observes whether the target produced anomalous output (e.g. a successful bootloader bypass), and updates its policy. ChipWhisperer's automated glitch explorer is an early implementation; LLM-guided fault hypothesis generation is unexplored.

Hardware Trojan Detection in Netlists

Hardware Trojans — malicious modifications inserted during IC manufacturing or design — are nearly impossible to detect by testing alone (they activate only under rare trigger conditions). ML applied to gate-level netlists can identify *structurally anomalous* subgraphs: clusters of gates with unusual fanin/fanout ratios, low observability, or activation patterns inconsistent with the module's stated function. GNNs trained on clean and Trojan-inserted netlists from the Trust-Hub benchmark achieve $>90\%$ detection on known Trojan families. Generalisation to novel insertion strategies is the open problem.

Microarchitectural Attack Surface Mapping

Spectre/Meltdown demonstrated that CPU microarchitectural features (speculative execution, branch predictors, cache hierarchies, TLBs) can be exploited without any software bug. Systematically discovering new microarchitectural vulnerabilities requires reasoning about CPU internal state that is not documented and not directly observable. ML approaches: training fuzzing feedback signals on micro-benchmark timing measurements to discover new cache-timing oracles; using LLMs to reason over CPU microarchitecture documentation and generate hypotheses about novel transient execution paths; and building ML models of branch predictor behaviour to identify new Spectre gadget patterns automatically.

Operating System and Kernel Security

Kernel Vulnerability Discovery via Static Analysis + ML

The Linux kernel codebase (>30 million lines) is one of the most security-critical and most complex codebases in existence. Traditional static analysers (Coccinelle, Sparse, Smatch, Coverity) produce thousands of alerts, most of which are false positives that developers ignore. ML applied here: learning from the history of real kernel CVEs which *patterns* of code are most likely to be exploitable (use-after-free via RCU races, integer overflows in `copy_from_user` paths, missing `capable()` checks before privileged operations). A classifier trained on (code pattern, CVE label) pairs from the kernel git history can rank Coverity alerts by exploitability, reducing analyst workload by an order of magnitude.

Syscall Sequence Anomaly Detection

Every process communicates with the kernel exclusively through system calls. The sequence and arguments of system calls encode what a process is doing. ML models (LSTM, Transformer, n-gram language models) trained on normal process syscall traces can detect anomalies in real time: a shell process that suddenly calls `ptrace` + `process_vm_readv` is likely performing process injection; a web server that calls `fork` + `execve` + `connect` in sequence has likely been exploited. Falco, Tetragon, and Sysdig use rule-based syscall monitoring; ML-based models can generalise beyond predefined rules and detect novel attack sequences never seen in training data.

AI-Assisted Kernel Exploit Development

The gap between "found a kernel bug" and "working privilege escalation exploit" is enormous and highly skill-dependent. It involves: understanding the kernel memory allocator (SLUB/SLAB) well enough to shape the heap; finding and chaining kernel gadgets for ROP; bypassing SMEP, SMAP, KPTI, and KASLR; and writing a reliable trigger sequence. LLMs have shown they can assist each sub-step when given sufficient context. The research question is how to build an *orchestrated pipeline* that takes a bug report (race condition in a network socket handler) and produces a tested privilege escalation proof-of-concept — with the LLM reasoning about kernel internals at each step and using symbolic execution to verify reachability.

OS Boot Chain Integrity Analysis

Modern boot chains have 4–6 layers: UEFI firmware, Secure Boot policy enforcement, bootloader (GRUB/systemd-boot), kernel image verification, initramfs, and early userspace. Each layer can be subverted if the layer below is compromised. ML can analyse UEFI firmware binaries (typically stripped PE32+ images) to identify: `SetVirtualAddressMap` callbacks that persist across OS reboot (a bootkit indicator); UEFI variables written outside of authenticated variable services (a persistence mechanism); and DXE/PEI driver dependencies that form an unusual execution graph relative to reference firmware. Tools like `uefi-firmware-parser` + LLM reasoning chains represent the current state of art; systematic ML-based anomaly detection on boot chains is essentially absent from the literature.

Memory Forensics and Runtime Analysis

Automated Memory Dump Triage with LLMs

When an incident responder captures a full memory dump of a compromised system, analysis today is largely manual: searching for injected shellcode patterns, finding unpacked malware images floating in heap memory, correlating process lists with network connections and loaded modules. LLM-augmented memory forensics (Volatility 3 + LLM MCP layer) can automate the triage: the LLM reads Volatility plugin output (pstree, netscan, malfind, dlllist), reasons about anomalies, forms hypotheses about the attack chain, and directs further targeted analysis. An SLM fine-tuned on (memory dump artifact, analyst annotation) pairs from real incident response cases is the missing piece.

Heap Spray and ROP Chain Detection in Memory

Live memory scanning for attacker artifacts — ROP chains, heap spray patterns, reflectively loaded PE images — is currently signature-based (`yara`, page-permission anomaly scanning). ML can go further: a CNN trained on memory page byte-distributions can classify pages as: code, data, heap, stack, mapped file, shellcode, ROP gadget pool, or heap spray target — without any signatures. This generalises to previously-unseen shellcode and packers that evade signature matching entirely.

Kernel Object Integrity Verification

DKOM (Direct Kernel Object Manipulation) attacks modify in-memory kernel data structures (e.g. unlinking a process from the `task_struct` doubly-linked list) to hide malicious processes from the OS's own process-enumeration APIs while keeping the process running. ML can detect DKOM by learning the *expected distribution of kernel object relationships* (number of threads per process, memory mappings per VMA, file descriptor table structure) and flagging statistical outliers. This requires no knowledge of the specific manipulation technique and generalises to novel rootkit implementations.

Firmware and Embedded Systems

Automated Firmware Unpacking and Analysis

Commercial firmware images are typically compressed, encrypted, or packed with proprietary formats that `binwalk` heuristics handle poorly. ML classifiers trained on entropy profiles and byte-frequency histograms can identify the compression/encryption format, locate the compressed payload, and select the appropriate decompressor. After extraction, LLMs can analyse the filesystem layout, identify init scripts, and flag obvious security misconfigurations (world-writable files in `/etc`, hardcoded credentials in `/etc/passwd`, `telnetd` enabled by default).

Bootloader Vulnerability Analysis

U-Boot, Barebox, and vendor-specific bootloaders are written in C, run with no MMU protection, and accept attacker-controlled input (TFTP downloads, USB mass storage, serial console). A single stack buffer overflow in a bootloader gives full control of the device before the OS even starts. LLM-assisted static analysis of bootloader source (or decompiled pseudocode for closed-source variants) with particular attention to: `memcpy`/`strcpy` on network-received buffers; environment variable parsing (often done with unsafe string operations); and USB descriptor processing (a historically rich attack surface).

RTOS Vulnerability Patterns

Real-Time Operating Systems (FreeRTOS, Zephyr, ThreadX, VxWorks, QNX) are present in medical devices, automotive ECUs, industrial controllers, and avionics. They lack most of the exploit mitigations present in Linux (ASLR, NX, stack canaries are optional or absent). ML trained on the CVE history of RTOS code (CVE-2021-31571 INTEGRITY RTOS, CVE-2020-25705 FreeRTOS+TCP) can identify recurring vulnerability patterns: integer overflows in packet length fields, missing bounds checks on task name buffers, and race conditions in scheduler queue manipulation. An LLM with RTOS-specific context (task priority semantics, ISR constraints, cooperative vs. preemptive scheduling implications for race windows) provides richer analysis than a general code model.

Cross-Binary TPM and Secure Enclave Interaction Analysis

Firmware that interacts with a TPM (Trusted Platform Module) or TEE (Trusted Execution Environment: ARM TrustZone, Intel TDX, AMD SEV) must follow strict protocols. Bugs in the non-trusted side's handling of TEE responses have led to exploitable vulnerabilities (e.g. checking the return code of a TEE

call but not the output buffer it populated). ML can learn the correct interaction patterns from well-audited reference implementations and flag deviations in new firmware — a form of specification mining applied to trusted-computing protocols.

AI-Augmented Exploit Primitives and Automation

Automated ROP Chain Compilation

Return-Oriented Programming requires finding and chaining *gadgets* — short sequences of instructions ending in `ret` — to build Turing-complete computation from existing code. Tools like ROPgadget, ropper, and pwntools automate gadget finding but require manual chain composition. ML approaches: training a seq2seq model on (desired computation, gadget chain) pairs extracted from CTF write-ups and exploit databases; using LLMs to reason about gadget semantics and compose chains that satisfy register constraints; and using RL to search the gadget graph when a greedy approach gets stuck.

Shellcode Polymorphism and Encoder Generation

Antivirus and EDR products detect shellcode by byte patterns. Encoders (shikata-ga-nai, alphanumeric encoding, UTF-8 encoding) transform shellcode into patterns that evade signatures while remaining functionally equivalent. Generative ML (GAN, VAE, or LLM-based) can produce novel encoders that simultaneously satisfy: functional equivalence (verified by an emulator), signature evasion (verified against the target AV), and size/alignment constraints. This is a research area that is actively studied by both offensive researchers and AV vendors developing defences.

Format String and Integer Overflow Auto-Exploitation

Format string vulnerabilities and integer overflows have well-understood exploitation patterns but require target-specific arithmetic (ASLR leak offsets, write-what-where target addresses, overflow trigger values). An LLM reasoning chain that: (1) identifies the vulnerable call site in pseudocode, (2) traces the data flow back to user input, (3) computes the necessary format string or integer values symbolically, and (4) produces a working PoC — is within reach given current LLM capability at code reasoning. The challenge is grounding the LLM's arithmetic in the actual binary's memory layout rather than hallucinated addresses.

Research Directions: Cybersecurity Reasoning Systems

A **Cybersecurity Reasoning System** (CRS) is a class of AI architecture that goes beyond pattern matching and retrieval to perform *multi-step, evidence-grounded, causally-aware inference* about security states, attacker intent, and defensive actions. The following research directions define the frontier of this discipline.

Taxonomy of Open Research Problems

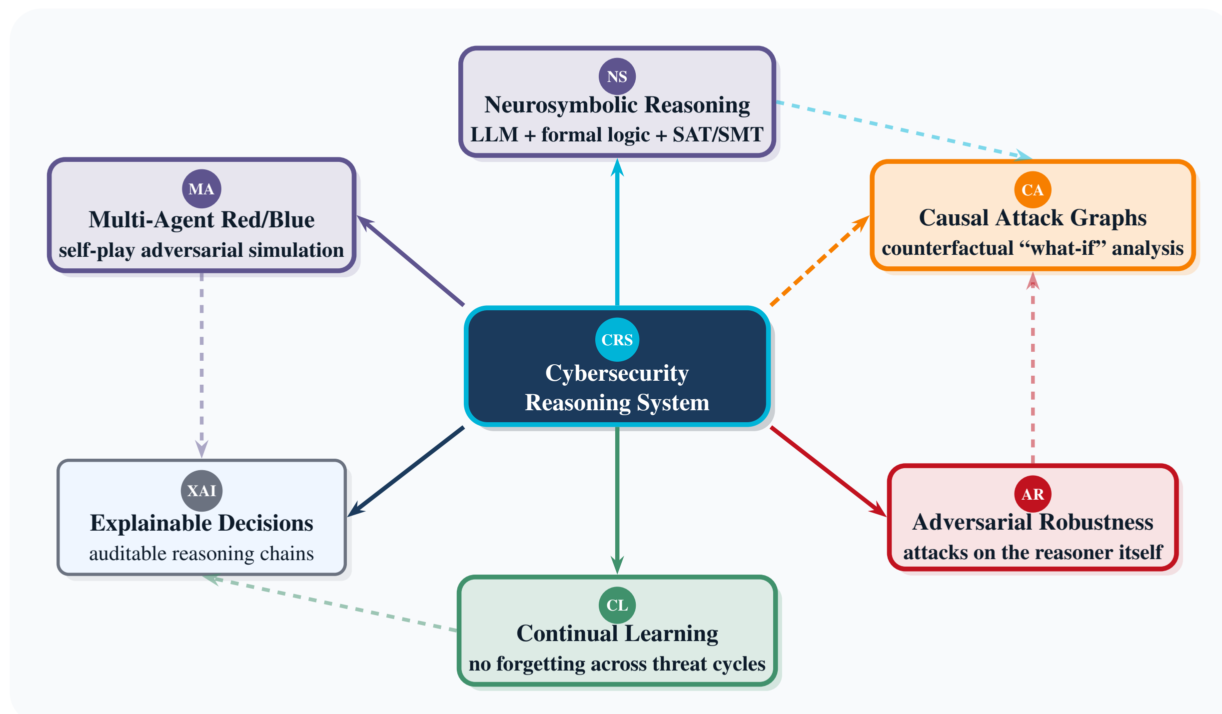

Figure 9. Six open research clusters around the Cybersecurity Reasoning System concept. Solid spokes show direct dependencies; dashed arcs show cross-cluster synergies.

Direction 1: Neurosymbolic Reasoning for Exploit Analysis

Core idea. Pure LLMs hallucinate on precise logical constraints (memory layout, register state, offset arithmetic). Pure symbolic solvers (Z3, Angr, Triton) are complete but require expert encoding. A neurosymbolic CRS uses the LLM as a *hypothesis generator* — proposing candidate exploit primitives in natural language — while a SAT/SMT backend *verifies* each candidate against the actual binary constraints. Only verified hypotheses advance to the next reasoning step.

Open problems.

- How to efficiently translate LLM-generated exploit hypotheses into SMT formulae without a human intermediary.
- Handling probabilistic constraints (heap spray success rates, ASLR entropy reduction) that pure SAT cannot express.
- Feedback loop design: how should a failed SMT check update the LLM's next hypothesis?

Practical starting point. Combine Angr (symbolic execution) with an LLM that reads the decompiled function and proposes vulnerability classes. Angr confirms or refutes; the LLM refines. Prototype on binary CTF challenges (e.g. pwn.college) to build a labelled dataset of (hypothesis, SMT-outcome) pairs for eventual fine-tuning.

Direction 2: Causal Attack Graph Reasoning

Core idea. Current attack graphs (MulVAL, CVSS chains) encode *logical* reachability but not *causal* counterfactuals: "If we patch CVE-2024-X, does the attacker still reach the crown jewel via an alternative path?" A CRS augmented with causal inference (Pearl's do-calculus, structural causal models) can answer counterfactual queries at scale, enabling prioritised remediation that accounts for path redundancy, not just individual CVSS scores.

Open problems.

- Building SCMs from heterogeneous sources (CVE data, network topology, IAM policies, log evidence).
- Real-time updating of the causal graph as the environment changes (new hosts, new vulnerabilities, lateral movement observed in SIEM).
- Uncertainty quantification: how confident is the model that a given attack path is *active* vs. merely possible?

Direction 3: Multi-Agent Red/Blue Adversarial Self-Play

Core idea. A Red Agent (attacker LLM + tools) and a Blue Agent (defender LLM + SIEM + patch automation) compete in a simulated environment. Neither agent has complete information. Both improve through play, analogous to AlphaZero's self-play in Go. Over thousands of episodes, the system discovers novel attack chains that human red teamers would take months to find, and defensive countermeasures that block them.

Architecture sketch.

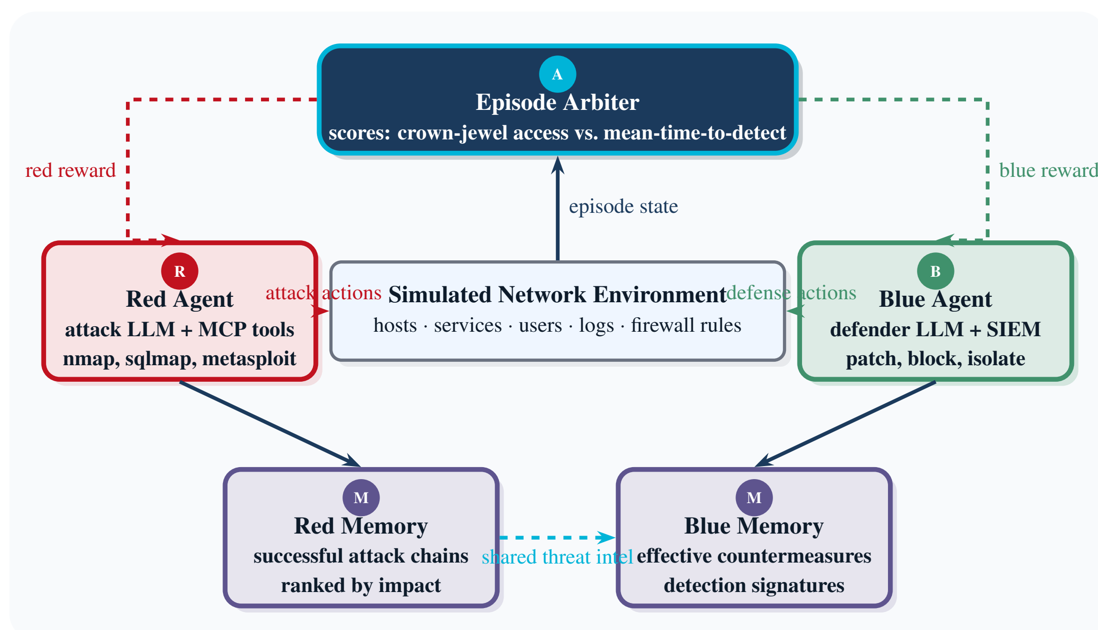

Figure 10. Multi-agent Red/Blue self-play architecture. Red and Blue agents act on a shared simulated environment; the arbiter scores each episode; both agents update their memory. After many episodes the Red memory becomes a novel threat-intelligence corpus and the Blue memory becomes a library of validated countermeasures.

Open problems.

- Realistic environment simulation without exposing production systems.
- Preventing the Red Agent from converging to trivially-exploitable simulation artefacts ("simulation overfitting") that do not transfer to real networks.
- Defining Blue reward functions that incentivise *early detection* rather than just eventual containment.

Direction 4: Continual Learning for Threat Detection

Malware families and attack techniques evolve continuously. A model trained on 2023 ransomware families will be blind to new variants introduced in 2025. **Continual learning** (also called lifelong learning) studies how to incorporate new knowledge without catastrophically forgetting old patterns — a core unsolved problem in neural network research that is especially acute in cybersecurity, where the class distribution

shifts rapidly and labelled data for new threats arrives slowly.

Promising approaches for security CRS.

- **Elastic Weight Consolidation (EWC):** penalises changes to weights important for previously-learned tasks when learning new ones.
- **Progressive Neural Networks:** add new model columns for new threat classes while keeping old columns frozen; lateral connections allow knowledge transfer.
- **Rehearsal with synthetic samples:** use a generative model to replay representative examples of old classes during new-class training.
- **Prompt-based continual learning for LLMs:** freeze model weights, add new knowledge via RAG and task-specific soft prompts, never backpropagate — the safest approach for production security deployments.

Direction 5: Explainable AI for Security Decisions

When an AI system flags a process as malicious, blocks a network connection, or assigns a CVSS score, a security analyst needs to understand *why* — both to validate the decision and to build institutional knowledge. This is not merely a regulatory requirement (GDPR Article 22 on automated decision-making, NIS2 audit obligations); it is operationally necessary because unexplained alerts are ignored.

Research priorities.

- **Faithful attribution:** SHAP and LIME produce post-hoc explanations that are often incorrect for deep models. Research into gradient-based attribution methods (Integrated Gradients, GradCAM for binary visualisation) that are *provably faithful* to the model's actual reasoning.
- **Reasoning chain externalisation:** training LLM-based detectors to emit a structured chain-of-thought alongside every classification ("flagged because: (1) three failed login attempts, (2) lateral move to finance subnet, (3) LSASS memory access within 60 s").
- **Counterfactual explanations:** "This alert would not have fired if the user had accessed the file from their usual subnet" — directly actionable for both analysts and policy authors.

Direction 6: Adversarial Robustness of the Reasoning System Itself

A CRS that can be manipulated by an adversary is worse than no CRS at all, because it creates a false sense of security. Known attack classes against ML-based security systems include:

- **Adversarial examples against malware classifiers:** adding benign-looking byte sequences to malware binaries that flip CNN/LSTM classifications from malicious to benign without altering execution behaviour (GAN-based evasion, Evadebert, MalGAN).
- **Data poisoning against intrusion detection:** injecting carefully crafted "normal" traffic into training data to carve out a blind spot for a specific attack pattern.
- **Prompt injection against LLM-based reasoning:** embedding adversarial instructions in scan output, log lines, or retrieved RAG documents that hijack the LLM's reasoning process (covered in §11.3 for MCP specifically).
- **Model extraction:** querying a deployed security classifier enough times to reconstruct a functional surrogate model, then using the surrogate to craft evasion samples offline.

Mitigations under active research. Certified defences (randomised smoothing), adversarial training with PGD-generated examples, input pre-processing (denoising, feature squeezing), ensemble disagreement detectors, and semantic consistency checks between the LLM's stated reasoning and its tool calls.

Research Agenda Summary

Direction	Readiness	Key open problem	First milestone
Neurosymbolic exploit reasoning	TRL 2–3	LLM→SMT translation	CTF binary solver
Causal attack graph reasoning	TRL 3–4	Real-time SCM updates	CVE prioritisation demo
Multi-agent Red/Blue self-play	TRL 2–3	Sim-to-real transfer	Capture-the-flag arena
Continual threat learning	TRL 4–5	Catastrophic forgetting	Malware stream benchmark
Explainable security decisions	TRL 4–5	Faithful attribution	SHAP audit for IDS
Adversarial robustness of CRS	TRL 3–4	Certified evasion defence	GAN evasion red-team
Pre-CVE vulnerability prediction	TRL 3	Labelled pre-patch data	GitHub commit classifier
Binary type recovery (stripped)	TRL 3–4	Cross-compiler generalisation	DIRTY benchmark replication
Deep learning side-channel SCA	TRL 4–5	Higher-order masking defeat	ASCAD masked key recovery
Kernel vuln classifier	TRL 3	CVE pattern generalisation	Coverity alert ranker
Syscall anomaly detection	TRL 4–5	Novel attack generalisation	Falco ML plugin PoC
Memory dump triage (LLM)	TRL 2–3	Incident annotation dataset	Volatility MCP layer
Heap spray / ROP CNN detector	TRL 3	Packer generalisation	Memory page classifier
Hardware Trojan GNN detection	TRL 3–4	Novel Trojan strategy	Trust-Hub SOTA replication
UEFI / boot chain anomaly ML	TRL 2	Reference firmware corpus	UEFI driver graph model
ROP chain auto-compiler	TRL 2–3	Gadget graph search	CTF pwn chain generator
Firmware auto-unpack + audit	TRL 3	Proprietary format coverage	IoT corpus classifier
Kernel exploit pipeline (LLM)	TRL 1–2	Kernel reasoning grounding	Race-condition PoC gen

Table 6. Research agenda: Technology Readiness Level (TRL) estimates and first milestone targets. TRL 1 = basic concept; TRL 9 = production deployment.

The Case for a Unified Cybersecurity Reasoning System

The six research directions above are not independent — they are facets of the same architectural challenge. A mature CRS must reason *causally* about attacker intent (Direction 2), *verify* its hypotheses formally (Direction 1), *explain* its conclusions to analysts (Direction 5), *adapt* to new threats without forgetting old ones (Direction 4), *resist* adversarial manipulation (Direction 6), and *improve* through adversarial self-play (Direction 3). No existing system does all six. This gap is the defining research challenge of AI-driven cybersecurity for the next decade. Teams that build modular, composable CRS architectures — where each direction is a swappable component rather than a monolithic system — will be best positioned to make incremental progress while preserving production deployability.

References & Further Reading

- Deng et al. (USENIX Security 2024) — *PentestGPT: An LLM-empowered Automatic Penetration Testing Framework.* https://github.com/GreyDGL/PentestGPT
- She, Pei et al. (IEEE S&P 2019) — *NEUZZ: Efficient Fuzzing with Neural Program Smoothing.* arXiv:1807.05620
- Nicolae, Eisele, Zeller (ESEC/FSE 2023) — *Revisiting Neural Program Smoothing for Fuzzing.* arXiv:2309.16618
- Su, Xu et al. (FSE 2024) — *CodeArt: Better Code Models by Attention Regularization When Symbols Are Lacking.* arXiv:2402.11842
- Luo et al. (NDSS 2023) — *VulHawk: Cross-architecture Vulnerability Detection with Entropy-based Binary Code Search.*
- Jiang et al. (ICSE 2024) — *BinaryAI: Binary Software Composition Analysis via Intelligent Binary Source Code Matching.* arXiv:2401.11161
- Aghaei et al. (2022) — *SecureBERT: A Domain-Specific Language Model for Cybersecurity.* arXiv:2510.00240

(v2.0)
- Invariant Labs (2025) — *MCP Security Notification: Tool Poisoning Attacks.* https://invariantlabs.ai/blog/
- OWASP — *MCP Security Cheat Sheet.* https://cheatsheetseries.owasp.org/cheatsheets/MCP_Security_Cheat_Sheet.html
- CyberArk (2025) — *Poison everywhere: No output from your MCP server is safe.*
- paddo.dev (2025) — *Running Claude Code Fully Local with Ollama.* https://paddo.dev/blog/claude-code-local-ollama/
- Google Security Blog (2025) — *Google announces Sec-Gemini v1.* https://security.googleblog.com/2025/04/
- Talos Intelligence (2025) — *Using LLMs as a reverse engineering sidekick.*