MirrorShield: Cross-Architecture Linux Malware Analysis via Lightweight Emulation and LLM

Zihan Zhang, Wenqi Lin, Lingna Sun, Hongyi Wang, Jingyi Wang, Yanshu Mei, and Fengwei Zhang*

Department of Computer Science and Engineering
Southern University of Science and Technology, Shenzhen, China
{12311124, 12311033, 12312418, 12313552, 12312713, 12412646}@mail.sustech.edu.cn
zhangfw@sustech.edu.cn

Abstract. Existing Linux malware analysis tools often struggle to scale across diverse CPU architectures, and many automated detectors yield only coarse-grained verdicts with limited forensic evidence. We present MirrorShield, a lightweight dynamic analysis framework that executes multi-architecture Linux binaries via containerized QEMU user-mode emulation and collects syscall-level telemetry via an eBPF-based kernel monitor. MirrorShield curates traces through local analyzers, denoising, and sampling, formatting them into structured prompts that produce evidence-backed reports. Across 11 CPU architectures, MirrorShield achieves 94.7% detection accuracy (F1: 0.946) with DeepSeek V3.2 and performs competitively across four LLM backends (91.6%–94.7%), confirming detection is driven by evidence rather than model choice. Detection stays above 90% even with 2,000× noise injection. An Evidence Match evaluation shows 81.4%–95.0% of key claims are directly supported by observed trace artifacts, confirming consistent grounding across all backends. MirrorShield also extracts structured behavior profiles, revealing family-specific signatures and shared operational patterns, while reducing turnaround time and overhead versus representative baselines.

Keywords: Malware Analysis · Dynamic Analysis · Cross-Architecture · Internet of Things · Large Language Models

1 Introduction

Malware volume and diversity have grown steadily over the past decade [1]. Although historically focused on Windows, the threat landscape has shifted significantly toward Linux, driven by IoT and embedded systems proliferation. Analyzing Linux malware, however, presents distinct challenges due to diverse CPU architectures (e.g., x86, MIPS, ARM) and specific runtime dependencies [2]. Static analysis is architecture-agnostic but undermined by statically linked binaries (over 80%) and packing that obscures semantics [2, 3]. Defenders therefore use dynamic analysis to recover runtime intent.

*Corresponding author.

Dynamic analysis can reveal concrete behaviors (e.g., filesystem, process, network, and persistence) hard to infer reliably from packed or obfuscated code. In practice, however, most malware sandboxes still rely on virtualization (VMs/emulation) to execute suspicious binaries [4, 5, 6, 7, 8, 9], creating a trade-off between scalability and transparency. Modern malware increasingly fingerprints virtualized or monitored environments and may suppress malicious actions [10, 11]. To mitigate artifacts, many systems move deeper into the stack via virtual machine introspection (VMI) or hardware-assisted analysis (e.g., PANDA, Ninja, MalT) [12, 13, 14]. These methods improve transparency but incur high overhead, limiting them to deep single-sample analysis. A lightweight, deployable framework scaling to large volumes while capturing meaningful behavior is therefore needed.

Automated detectors also trade interpretability for accuracy: rule-based systems generalize poorly, and black-box ML models offer limited forensic insight. LLMs can transform low-level traces into interpretable forensic reports. However, the sparsity and volume of execution logs often overwhelm LLM context windows, leading to missed signals [15] or unsupported details. This requires explicit evidence-aware summarization and grounding [16] to ensure reliable analysis within tight prompt budgets.

We introduce MirrorShield, an evidence-grounded framework pairing lightweight emulation with kernel-level telemetry and LLM reporting. To handle limited LLM budgets, we denoise and sample traces with alert guidance and feed the resulting core syscall evidence through a structured prompt. Evaluation across 11 CPU architectures shows MirrorShield achieves 94.7% detection accuracy (F1: 0.946) while generalizing well across architectures. We further validate the pipeline across four LLM backends under identical evidence and prompts, finding accuracy spans 91.6%–94.7% and different models exhibit distinct precision–recall trade-offs, confirming robustness to backend choice. In terms of efficiency, the pipeline adds only modest runtime overhead ($<10\times$) and is substantially faster than representative system-level sandboxing baselines, where slowdowns can exceed 2,000×. To assess robustness under noisy traces, we conduct extreme context flooding and find that detection remains above 90% even with up to 2,000× noise injection [17, 18], with added cost saturating as noise increases. To quantify report reliability, we introduce an *Evidence Match* evaluation, finding 81.4%–95.0% of key evidence claims directly link to observable trace artifacts across all evaluated backends. MirrorShield's reports also support structured behavior profiling, revealing consistent family fingerprints and a shared operational skeleton across 844 malware samples. Our contributions are:

- We build MirrorShield using containerized QEMU user-mode execution and eBPF-based syscall tracing to support 11 CPU architectures for scalable end-to-end malware analysis on commodity x86 hosts. We open-source MirrorShield at a repository alongside our evaluation data.
- Our proposed trace curation pipeline combines denoising, frequency-based sampling, and alert-guided prioritization to fit execution evidence into tight LLM context budgets, maintaining high accuracy under heavy syscall pollution.

- Evaluating MirrorShield across four LLM backends with identical structured evidence and prompts reveals consistently strong performance (91.6%–94.7%) while characterizing the precision–recall trade-off distinguishing different model behaviors.
- We introduce an Evidence Match metric assessing report grounding at the trace level. Results show observed runtime artifacts directly support 81.4%–95.0% of key claims, ensuring report auditability and reproducibility across all backends.
- Behavior profile extraction from LLM-generated reports uncovers family-discriminative signatures and a stable core triad shared by most Linux malware samples.

2 Related Work and Assumptions

2.1 Dynamic Malware Analysis Platforms

Agent-Hook-Based Sandboxes. Early platforms relied on in-guest agents to intercept system calls and APIs. Cuckoo hooks APIs via in-guest components; LIMON wraps `strace` to trace execution [4, 5]. Other works for IoT threats, such as Detux and LISA, focused on capturing network traffic or provided multi-architecture sandboxes [6, 7]. Commercial solutions like Joe Sandbox also offer cloud-based cross-platform capabilities [19]. Their monitors share the malware's privilege domain, introducing detectable artifacts and enabling evasion.

VMI and Hardware-Assisted Approaches. To address evasion, later work moved monitoring outside the guest OS via VMI. DRAKVUF uses Xen and LibVMI to transparently track system calls for rootkit analysis [20]. PANDA provides record-and-replay capabilities with deep taint analysis [12] and PyREBox enables scriptable instrumentation of running systems [21]. Similarly, commercial solutions like VMRay Analyzer use hypervisor-level monitoring to remain stealthy [22]. Ninja and MalT use ARM TrustZone or System Management Mode to hide the monitor [13, 14]. Despite improved stealth, PANDA's record-and-replay introduces roughly a 4× slowdown over a baseline VM run, and heavyweight plugins can push overhead up to 100× [12].

Container-Based Isolation. gVisor provides lightweight isolation, but its user-space syscall interception may conceal actual malware behaviors [23, 24, 25]. Existing eBPF tools like Falco target runtime defense, not granular forensic introspection [26].

2.2 Automated Detection and Interpretation

Machine Learning-Based Detection. Machine Learning (ML) approaches automate malware classification based on extracted features. Works like EMBER utilize static features from PE files or raw bytes to train classifiers [27]. For dynamic behaviors, Xiao et al. applied LSTM networks to model system call sequences in Android malware [28]. Deng et al. proposed Enimanal for IoT malware, which converts system-call traces into an attributed system-call graph and

Table 1. Comparison of representative Linux malware analysis systems across dynamic analysis capabilities.

System	Cross-Arch†	Emulation	Telemetry	Analysis	Output
LiSa [7]	Yes	QEMU	SystemTap	None	Logs/summary
Limon [5]	No	VM	`strace`	None	Logs and artifacts
Enimanal [29]	Yes	QEMU	`strace`	ASCG and GNN	Label/score
ELFEN [9]	No	QEMU	eBPF	Behavior rules	Score and logs
MirrorShield	**Yes**	Container and QEMU-user	eBPF (with cgroup)	Rules and LLM	Evidence and explanation

†Cross-Arch denotes single-host cross-architecture execution and analysis.

applies GNN-based classification, improving robustness by augmenting node/edge semantics using system-call specifications [29]. Despite their effectiveness, these black-box models offer limited explainability and generalize poorly to zero-day exploits and adversarial examples [30].

LLM-Assisted Analysis. LLMs have been applied to semantic malware analysis, including deobfuscation (e.g., Androidmeda [31]) and TTP generation [32], as further evidenced by benchmarking frameworks for Code LLMs on Android malware [33]. However, their use in dynamic analysis remains limited. Prior studies show LLMs can suffer from hallucinations and shallow reasoning when analyzing complex code [34]. Moreover, finite context windows and high inference costs make feeding raw execution traces directly into LLMs impractical [35]. Our work pairs lightweight kernel-level monitoring with LLM reasoning while explicitly bounding data volume.

2.3 Scope and Assumptions

We study scalable cross-architecture Linux malware analysis using syscall-level telemetry, deriving behavior-related signals from syscall names and arguments. We consider adversarial inputs flooding traces with noise or exploiting limited LLM context budgets [17, 18]. To reduce reliance on user space tracing easier to detect and disrupt [36, 10, 11], we use kernel-side eBPF telemetry instead. We assume the host kernel and monitoring stack are trusted. Host compromise is out of scope, including container escape, host-kernel exploits, and kernel rootkits tampering with telemetry.

3 System Design and Implementation

Figure 1 illustrates the high-level architecture of MirrorShield. The controller runs cross-architecture binaries concurrently in Docker containers via QEMU-USER emulation (Section 3.1). The kernel layer captures runtime behaviors via eBPF with Cgroup-based filtering for low-noise, high-speed collection (Section 3.2). The stream is routed to the analysis engine for the corresponding sample (Section 3.3). A local analyzer profiles behavior and samples traces to fit LLM context limits. The retained signals and container output are formatted into a structured prompt and sent to an LLM for analysis.

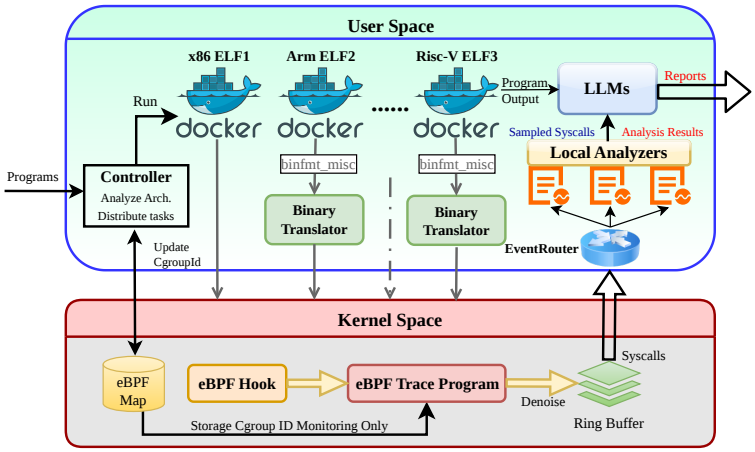

Fig. 1. Overall System Design of MirrorShield

3.1 Execution Workflow and Environment

MirrorShield begins execution by traversing the target directory. It parses each binary's ELF header to identify its Instruction Set Architecture (ISA) and library dependencies, matching them with the appropriate Docker image from our official set. The controller then spawns containers in a dormant state and registers their Cgroup IDs into a BPF map to enable container-specific event tracing. Once the environment is prepared, the target binary is injected via `docker cp` and executed using `docker exec`. Upon termination or timeout, a cleanup routine kills processes and removes containers to release resources.

Isolation Configuration. We configure the execution environment to use Docker's native isolation capabilities, as summarized in Table 2. We enable the User Namespace feature [37], mapping the container root to an unprivileged host user. In practice, over 83% of containers run with host-root privileges [38] due to missing user isolation in default runtimes. This remapping prevents privilege escalation from reaching the host kernel while restricting container capabilities [39]. We also use Cgroups to enforce resource quotas, limiting CPU and memory to prevent exhaustion.

Host-Resident Interpreter Injection. As illustrated in Figure 2, to execute cross-architecture binaries without modifying container images, we leverage the Linux kernel's `binfmt_misc` mechanism [40] together with FD pinning to bind the host's static `qemu-user` binary to executable file signatures. This mechanism allows the kernel to transparently redirect `execve` of non-native binaries to QEMU user-mode inside the container's namespace. QEMU user-mode performs instruction-set and system-call translation without virtualizing hardware or

Table 2. Isolation Configuration Overview

Mechanism	Isolation Capability / Policy
Default Docker Isolation	
PID/Net/IPC Namespaces	Isolate process view, network, and signals
Mount Namespace	Provide independent filesystem view
UnionFS / OverlayFS	Copy-on-Write filesystem
Enhanced Security Configuration	
User Namespace (UserNS)	Remap container root to host non-root
Cgroups	Enforce CPU/Memory limits; Disable Swap
Storage Strategy	No host volumes mounted

running a guest OS [41], while the container supplies the target architecture's user space environment (e.g., dynamic linker, libc, and shared libraries) and namespace-based isolation. Unlike QEMU system emulation, which requires full hardware virtualization and a guest OS, our design composes lightweight ISA translation with containerized user space environments, avoiding VM-level overhead while enabling scalable, container-native execution of heterogeneous binaries via the kernel's standard process launch path.

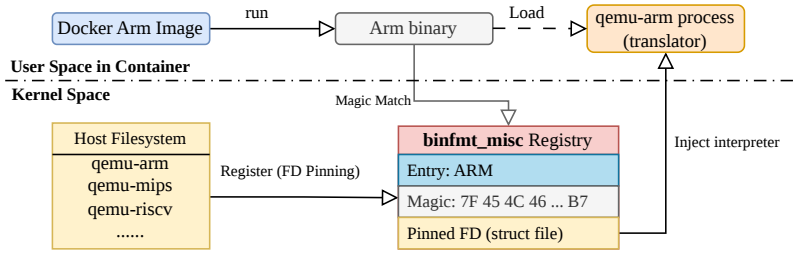

Fig. 2. FD-pinned binfmt interpreter for cross-architecture execution

3.2 Kernel Space Observability

The eBPF-based monitor provides kernel-level visibility and runs across kernel versions without recompilation.

Table 3 lists our five behavioral categories, covering fileless execution (`memfd_create`), debugger detection (`ptrace`), and other security-critical calls while filtering system noise. Beyond syscall identifiers, our probes extract selected arguments defining the semantic context of each call (e.g., file paths, flags, PIDs, and namespace or memory attributes).

Dynamic Cgroup-based Filtering. System-wide hooks would introduce unacceptable overhead and noise, so we filter events to analysis containers only, without in-guest agents. The filtering logic relies on synchronization between

Table 3. Monitored Kernel Events and Behavioral Semantics

Category	Key Tracepoints	Security Semantics
Process Lifecycle	`execve, clone, sched_*, sched_process_*`	Execution tree reconstruction; Shellcode spawning inheritance.
FileSystem	`getdents64, readlinkat, openat, access, sys_enter_*`	Environment fingerprinting; Persistence checks; Config scanning.
Network Activity	`connect, bind, sendto, ip_*, recvfrom, inet_*`	C2 signaling; Data exfiltration; DGA domain resolution.
Memory	`memfd_*, vma_*, mm_page_*, process_vm_writev, mprotect`	Fileless execution; Packer unpacking; Remote code injection.
Evasion & Security	`ptrace, prctl, seccomp, sys_enter_bpf, security_*`	Anti-debugging checks; Sandbox evasion; Exploit attempts.

the user space controller and the kernel space BPF program. As containers are spawned, MirrorShield extracts their Cgroup IDs and registers them into a BPF Hash Map (`monitored_cgroups`). The following code shows our main logic in Algorithm 1. Upon capturing a system call, the in-kernel eBPF probe applies filtering logic and caches the target process PID to optimize subsequent lookups.

Algorithm 1: Dynamic Cgroup Filtering Logic

```
Data: monitored_cgroups: Hash map ⟨CgroupID → 𝔹⟩ storing active containers;
monitored: Hash map ⟨PID → 𝔹⟩ as PID cache;
Input: Current process identifier pid
Output: Boolean indicating whether process should be monitored
Function should_monitor(pid):
    if pid ∈ monitored then
        return true;
    cg ← bpf_get_current_cgroup_id();
    if cg ∈ monitored_cgroups then
        monitored[pid] ← 1 ;                          // Update PID cache
        return true;
    return false;
```

In-Kernel Noise Reduction. Benign system calls (e.g., library loading, memory allocations) dominate traces and degrade performance. We aggregate repetitive events in-kernel to filter redundancy at the source. We utilize two distinct aggregation strategies. First, for file system noise, we implement categorization logic (`categorize_path`) to identify non-malicious activities. Access to artifacts like shared libraries (`.so`) is intercepted but not submitted to the ring buffer; instead, we increment counters in a `BPF_MAP_TYPE_ARRAY` (`agg_counters`). Second, for high-frequency memory operations, we employ statistical summarization. Rather than logging every `mmap` syscall, we track total execution count and maximum single allocation size in a dedicated map (`mmap_stats`). These aggregated statistics are flushed to user space periodically, identifying potential large payload injections while avoiding log saturation.

3.3 AI-Assisted Behavioral Analysis

We process raw kernel events through a two-stage pipeline (Figure 3). Raw system calls flow along two parallel paths: one stream is aggregated for local analyzers to identify and alert on suspicious activities, while the other undergoes denoising before being fed directly into the LLM. The LLM acts as the central reasoning engine, integrating these local alerts and denoised syscalls with specific prompts and raw binary output to generate a final, detailed report.

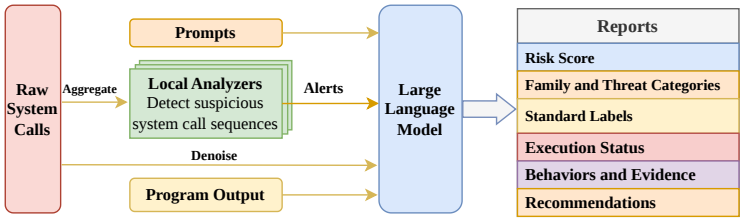

Fig. 3. AI-Assisted Two-Stage Behavioral Analysis Architecture

Data Aggregation and Rule-Based Detection A sliding-window algorithm compresses repetitive syscalls prior to LLM invocation to preserve the underlying execution logic. The compressed stream feeds 20 lightweight analyzers that apply rules extracted from common Linux malware chains (reconnaissance → privilege escalation → persistence). Argument-level constraints, such as specific `mprotect` flags and sensitive `mount` targets, suppress false positives. An `AnalyzerManager` tracks per-sample state to capture multi-step patterns as temporal correlations. Monitored patterns include `connect` followed by `dup2` redirecting sockets to standard I/O (`ReverseShell`), `process_vm_writev` after prior `ptrace` activity (`ProcessInjection`), and fork bursts within a 2-second window (`ForkBomb`). Each alert assigns a severity-based threat score to guide subsequent LLM evaluation at the parameter level. Table 4 lists two representative analyzers per behavioral category; the remaining eight (e.g., `MemoryCorruption`, `SandboxBypass`, `LogTampering`, `InfoLeakage`) follow an identical design.

Trace Denoising and Context Construction. If the trace still exceeds the LLM context window after aggregation, we apply priority sampling before submission. For logs exceeding the token limit, we employ a frequency-based sampling algorithm. This method strictly preserves rare events and head/tail context (startup/shutdown sequences) while statistically downsampling high-frequency background noise. The final LLM context combines three sources: (i) the denoised syscall trace, (ii) local analysis alerts, and (iii) raw binary output. We then wrap the merged context into a structured prompt for the final reasoning stage.

Table 4. Representative Analyzer Detection Logic.

Category	Analyzer	Detection Logic & Constraints
Memory & Code Injection	FilelessExec ProcessInjection	`exec /proc/*/fd/*` / `memfd_create` `process_vm_writev` + `memfd/ptrace`
Container & Kernel	ContainerEscape KernelManip	`setns/pivot_root` / `mount(proc,sysfs,host)` `init_module/finit_module` / `bpf(cmd=5)`
Persistence & File System	Persistence SSHBackdoor	18 patterns: `ld.so.preload`, `systemd`, `cron`, profiles write `.ssh/authorized_keys`, `sshd_config`
Network & Exfiltration	ReverseShell RawSocketSniff	`connect` → `dup2(fd,{0,1,2})`; per-PID state `socket(AF_PACKET, SOCK_RAW, ETH_P_ALL)`
Resource & Exploitation	ForkBomb RaceCondition	`fork`/`clone` > 50 in 2 s window Dirty COW: `madvise(DONTNEED)` ≥ 5 + write(`/proc/self/mem`)
Access Control	AccessControl SuidSgidPermChg	`setuid(0)` / path traversal / `/etc/shadow` access `chmod(SUID/SGID)` on `/bin/*`, `/sbin/*`

LLM-Driven Reasoning. The LLM receives a fully structured prompt merging the denoised syscall trace with deterministic local alerts. We instruct the model to act as a `Malware Analysis Expert` and to ground all conclusions in observable evidence. Given the limited context window, local alerts serve as high-confidence anchors pointing to security-relevant behaviors, even when some background syscalls are downsampled. To avoid over-constraining the model, analyzers emit conservative alerts providing guidance rather than rigid rules. The prompt also injects lightweight domain priors, including family-specific behavioral signatures (Appendix B), treated as hypotheses that must be validated against the trace and binary output. Accordingly, the LLM performs three tasks: (i) verify and contextualize alerts by correlating them with nearby trace events and output, (ii) reconstruct intent from fragmented low-level actions, and (iii) assign a verdict and risk score (0–100) and produce a standardized report with threat categories, execution status, and mitigation recommendations.

4 Evaluation

All experiments were conducted on an ASUS Zenbook 14 Air equipped with an Intel Core Ultra 7 258V processor, 32GB of RAM, and a 1TB SSD. The system runs Ubuntu 24.04 LTS with Linux kernel 6.8.10. We evaluate MirrorShield end-to-end around five questions.

RQ1. What is the classification accuracy of MirrorShield, and how does the choice of LLM backend affect performance (Sec. 4.1)?
RQ2. How broad is MirrorShield's architecture coverage, and does it maintain accuracy across diverse ISAs and noisy traces (Sec. 4.2)?
RQ3. How accurate and reliable are the generated report conclusions (Sec. 4.3)?
RQ4. Is MirrorShield efficient enough for high-throughput analysis compared to standard Linux sandboxes (Sec. 4.4)?

RQ5. Can MirrorShield extract consistent, family-discriminative behavior profiles at scale (Sec. 4.5)?

Dataset Construction. We collected 844 malware samples from MalwareBazaar [42], selecting samples across 10 threat signatures and cross-referencing their `SHA-256` hashes with VirusTotal [43] for family classifications. As shown in Fig. 4, the dataset covers nearly all prevalent CPU architectures and spans a broad range of malicious behaviors, mostly botnet-related. We also collected 844 benign ELF binaries. These samples span 7 distinct CPU architectures, ensuring the benign dataset mirrors the architectural heterogeneity of our malware collection. [1]

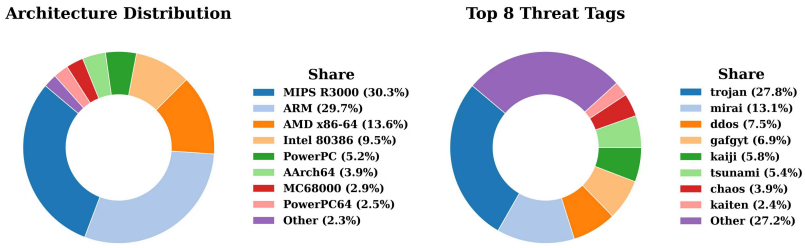

Fig. 4. Distribution of the collected malware dataset. The right chart depicts the top threat tags, while the left chart shows the targeted CPU architectures.

4.1 Detection Effectiveness

Overall Accuracy. Using DeepSeek V3.2 [44] as the default analyzer, the pipeline achieves 94.7% accuracy with a 0.946 F1 score (Table 5(a)). Performance correlates with execution stability: cleanly executing samples reach 96.17%, while the roughly 12% with crashes or dependency failures drop to 90.95% (Table 5(b)). The main outliers are *Ransomware* and *Hailbot*, whose dormant behaviors require specific environmental triggers, and benign administrative utilities such as *netcat* and *curl*, whose sensitive-looking operations occasionally trigger false positives.

Multi-LLM Sensitivity. To assess generalization, we evaluate four backends under identical structured evidence and prompts (Table 5(c)). All four models achieve competitive accuracy, confirming that detection quality is driven by the evidence pipeline rather than the specific LLM. A clear behavioral split emerges: DeepSeek and Qwen symmetrically balance false rejections and negatives, while OpenAI and Gemini act conservatively, producing markedly fewer false rejections but missing more malware. Neither is universally preferable; choice depends on prioritizing minimal false alarms or maximal recall.

[1] The benign dataset includes samples sourced from public GitHub repositories: Labeled-Elfs, mips-binaries, riscv-tests-prebuilt-binaries and linux-static-binaries.

Table 5. Detection Breakdown, Architecture Support, and Multi-LLM Comparison. **(a)** Acc._{DS} and Acc._{Q} denote DeepSeek V3.2 and Qwen Plus. **(b)** Acc. is based on DeepSeek V3.2; Exec. Err means non-zero exit, crash, or timeout. **(c)** All backends use identical evidence and prompts.

(a) Detection Performance

Class/Family	Total	Acc._{DS}	Acc._{Q}
CoinMiner	35	91.4%	97.1%
Gafgyt	156	94.9%	94.2%
Hailbot	35	85.7%	100%
Kaiji	160	97.5%	98.8%
Mirai	148	93.9%	95.9%
Mozi	42	100%	40.5%
Tsunami	160	94.4%	95.6%
XorDDoS	73	97.3%	98.6%
Docker Exp.	12	100%	100%
Ransomware	23	73.9%	52.2%
Malware Sub.	**844**	**94.5%**	**92.7%**
Benign Sub.	**844**	**94.8%**	**90.5%**
Global Total	**1688**	**94.7%**	**91.6%**

(b) Arch. Coverage & Stability

Arch	Count	Acc.	Exec. Err
x86 Family	563	94.5%	12.6%
ARM Family	696	92.4%	11.5%
MIPS	267	94.4%	18.4%
RISC-V	28	100%	17.9%
PowerPC	56	89.3%	17.9%
SPARC	20	100%	5.0%
SuperH	33	97.0%	3.0%
m68k	25	100%	0.0%
Global	**1688**	**94.7%**	**12.9%**

(c) Multi-LLM Comparison

Analyzer	ACC	F1_{mal}	FRR	FNR
DeepSeek V3.2	94.7%	0.946	5.2%	5.5%
Qwen Plus	91.6%	0.917	9.5%	7.3%
OpenAI GPT-5.2	92.0%	0.916	3.4%	12.7%
Gemini 3.1 Pro	93.5%	0.931	2.1%	11.0%

Component Contribution. To isolate the impact of sampling and alert grounding under a fixed trace budget, we control the number of syscall events seen by the analyzer. We construct a *no-sampling* variant that takes the first n syscalls, and a *no-alert* variant that withholds analyzer alerts from the LLM prompt. Alerts are intentionally conservative, triggering on only 6.60% of benign reports and thereby preserving a favorable signal-to-noise ratio. Removing either component degrades malicious-only detection to roughly 85%; enabling both recovers the full 94.7% accuracy, demonstrating that sampling and alert grounding are complementary and jointly necessary.

4.2 Cross-Architecture Generalization and Robustness

Architecture Coverage. Unlike existing sandboxes limited to x86 or specific ARM versions, our system covers a broad range of ISAs. Mainstream architectures show great robustness, achieving average detection rates exceeding 93%. Table 6 summarizes our architecture support, extending beyond mainstream ISAs to emerging targets like RISC-V. For long-tail architectures observed in the wild (e.g., SPARC, SuperH, and m68k), we execute samples using our default x86_64 container environment, extracting behavior-relevant indicators from the resulting traces. Across these cases, we observe consistently high precision even without a dedicated Docker base image for the target ISA.

Architecture Stability. Table 5(b) evaluates cross-architecture robustness. While mainstream ISAs (e.g., x86 and ARM) execute stably, MIPS and RISC-

Table 6. Architecture support comparison. † Covers 32/64-bit and BE/LE variants where applicable: `i386/x86_64`; `arm/aarch64` (BE/LE); `mips/mipsel`; `mips64/mips64el`. ‡ Covers `ppc64` and `ppc64le` (evaluation mainly on `ppc64le`).

Architecture	Cuckoo [4]	Lisa [7]	Detux [6]	Limon [5]	Triage [45]	Any.Run [8]	ELFEN [9]	**Ours**
x86	✓	✓	✓	✓	✓	✓	✓	✓†
ARM	✓	✓	✓	–	✓	✓	32bit	✓†
MIPS	–	✓	✓	–	✓	–	✓	✓†
RISC-V	–	–	–	–	–	–	–	✓
PowerPC	–	–	–	–	–	–	✓	✓‡
s390x	–	–	–	–	–	–	–	✓

V exhibit higher failure rates (up to ∼18%). These failures stem mainly from sample–environment mismatches under emulation, including missing dependencies, ABI/loader differences, and brittle runtime assumptions. Many MIPS samples are also legacy IoT artifacts with aging software stacks incompatible with modern runtime environments. Because the eBPF probe captures the syscall sequence preceding a crash, detection accuracy holds despite execution instability.

Robustness to Context Flooding. We evaluate context flooding robustness by stress-testing 50 randomly sampled malware traces with injected benign noise. For each malicious system call, we append λ benign events drawn from a small pool of neutral patterns (e.g., opening `/dev/null`), and assign them the same PID with timestamps adjusted to preserve temporal consistency. Varying the injection ratio λ from 5 to 2000 increases the raw trace length by up to 2000×. Figure 5 shows that performance degrades gracefully under heavy noise: accuracy remains 98% at λ=1000 on this set, and stays above 90% at λ=2000. We also observe input-size saturation: although the raw trace grows linearly with λ, the LLM input stabilizes at roughly 19k tokens for $\lambda \geq 50$. The local analyzer thus enforces a fixed processing budget, substantially limiting context flooding impact.

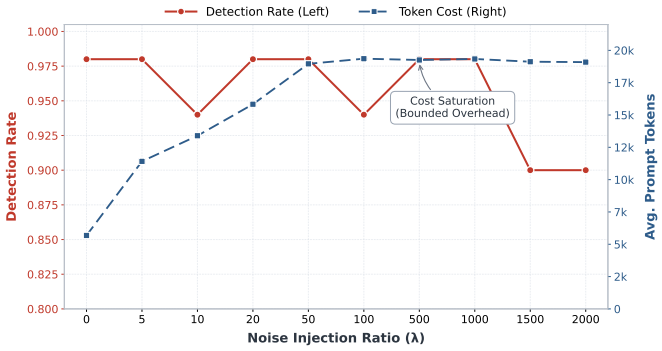

Fig. 5. Robustness under Extreme Noise Injection. Despite noise increasing linearly up to 2000×, Token Cost (blue dashed) plateaus at $\approx 19k$ (bounded overhead), while Detection Rate (red solid) stays above 90%.

4.3 Explanation Quality and Grounding

VirusTotal Consistency Check. We use a VirusTotal (VT) [43] consistency check for external threat intelligence validation. Specifically, we extract the predicted family and `threat_category` from each LLM-generated report, comparing them against a reference token pool built from VT's `suggested_threat_label` and `popular threat names`. After token normalization, we report four metrics: (i) family match, (ii) category match, (iii) any match (family or category), and (iv) average category match ratio (Table 7). Results reveal consistent trends across all four analyzers. DeepSeek achieves the highest category match (90.1%), followed by Gemini (87.1%) and Qwen (74.7%); for family matching, the rates are 55.6%, 44.7%, and 59.0% and 31.2% respectively, with OpenAI exhibiting the lowest family-token overlap. OpenAI shows a markedly lower any-match rate of 66.1%, attributed to its tendency toward more abstract or conservative threat vocabulary rather than incorrect behavioral understanding. Overall, category-level agreement consistently exceeds fine-grained family token overlap across all models, an expected pattern given VT aliasing and heterogeneous naming conventions. We therefore treat this check as evidence of capturing correct high-level behavior, rather than as strict ground-truth family labeling.

Table 7. Grounding results (VT consistency and Evidence Match). VT rates are based on 793 (DeepSeek/Qwen) or 854 (OpenAI/Gemini) matched reports; SR_{all} is based on 844 reports with traces.

Analyzer	Family	Category	Any	Avg. ratio	$\mathbf{SR_{all}}$
DeepSeek V3.2	55.6%	90.1%	93.8%	48.2%	81.4%
Qwen Plus	59.0%	74.7%	88.9%	33.4%	89.4%
OpenAI GPT-5.2	31.2%	66.0%	66.1%	40.9%	90.0%
Gemini 3.1 Pro	44.7%	87.1%	87.1%	43.3%	95.0%

Evidence Match. Because external labels can be noisy, we further introduce *Evidence Match* to measure trace-level grounding: verifying whether each `Key Evidence` item in a report exists in the corresponding raw JSONL execution trace. For each report, we normalize every evidence bullet into matchable indicators (Table 8), and perform line-level matching against the trace. An evidence item is marked as *supported* if any extracted indicator appears in the trace, and *unsupported* otherwise. Let E_r be the number of evidence items in report r, and S_r the number of supported items; the overall micro-averaged support rate is $\mathrm{SR}_{\mathrm{all}} = \sum_{r\in\mathcal{R}} S_r \,/\, \sum_{r\in\mathcal{R}} E_r$. Across 844 reports, support rates rise monotonically from DeepSeek ($\mathrm{SR}_{\mathrm{all}} = 81.4\%$) to Qwen (89.4%), OpenAI (90.0%), and Gemini, achieving the highest trace grounding at 95.0% (Table 7). This ordering inversely mirrors the VT any-match: OpenAI and Gemini produce fewer VT family-label hits but ground their evidence more tightly in the raw trace. This suggests that Evidence Match and VT consistency provide complementary perspectives, as models might use different threat vocabulary than VT while still anchoring key claims to observable runtime artifacts. Ultimately, these results confirm that most reported evidence, regardless of the LLM, directly traces back to execution telemetry, ensuring generated reports remain reproducible and easily auditable.

Table 8. Indicator extraction and matching rules in Evidence Match.

Indicator	Rule
Path token	Extract `/..` substrings (e.g., `/tmp/..`); supported if any hit appears in the trace.
Uppercase marker	Extract uppercase/underscore tokens (e.g., `CONNECT`); supported if any hit appears in the trace.
IPv4	Extract IPv4; supported on direct/decoded hits; also decode common hex-encoded IPv4 from trace lines.
Fallback word	Otherwise extract alphabetic words ($\geq$4 chars); match case-insensitively in the trace.

4.4 Performance and Scalability

Configuration and Setup. We performed experiments on a host machine running our system with 7 concurrent Docker containers. For each container, we enforced strict Linux cgroups limits: a 1 GB memory cap, a 30% CPU quota, 1,000 maximum processes (PIDs), 50 fork operations limit, and a 60-second hard execution timeout. We deployed a full system emulation via `qemu-system-aarch64`. The virtual machine featured a Cortex-A57 vCPU, 512 MB RAM, and Alpine Linux (kernel `vmlinuz-virt`). The emulation ran in non-graphic mode with standard VirtIO networking.

System Call Overhead. We quantify syscall monitoring cost using LMbench `lat_syscall`, which repeatedly invokes a minimal system call in a tight loop and reports the steady-state per-call latency [46]. A common approach for cross-architecture sandboxing is to run samples under `QEMU` emulation and recover syscall-level visibility via `strace` [47, 29]. We thus adopt two representative baselines: (i) native `strace` as a lightweight user space tracer, and (ii) `QEMU` user-mode with `strace` as a standard cross-architecture tracing pipeline. We further include two recent research systems as references. LiSa combines QEMU-based execution with SystemTap to collect behavioral telemetry [7], while the QEMU-based ELFEN primarily uses eBPF for runtime telemetry [9]. As summarized in Table 9(a), MirrorShield introduces only single-digit syscall overhead across architectures (4.4×–9.6×). In comparison, ptrace-based tracing is typically hundreds to thousands of times more expensive. LiSa and ELFEN also show higher overhead, as heavier monitoring and full-system emulation incur substantially higher cost.

Lifecycle Agility (Startup & Teardown). We evaluated the system's agility by benchmarking the effective startup and teardown latencies against LiSa [7]. For a fair comparison, we instrumented both systems' orchestration logic to capture precise timestamps. For MirrorShield, startup spans from Docker provisioning initialization to the eBPF probe receiving the first system call event. For the baseline, we modified the LiSa source code to log internal state transitions, defining startup as the interval from QEMU VM spawning to the `staprun` instrumentation loader's completion. Teardown latency spans from the target program's termination (or timeout) to complete container or VM shutdown. Measurements

were averaged across distinct trials using diverse samples per architecture to account for initialization variance. Eliminating full-system emulation overhead yields a 4.8× to 14.6× speedup across architectures (Table 9(b)).

Table 9. Performance Breakdown: System Call Overhead (Left) and MirrorShield Lifecycle Latency (Right).

(a) System Call Latency & Overhead

Environment	Lat. (ms)	Overhead
Native Host (x86)	0.049	1.00×
Native `strace`	16.01	327×
MirrorShield		
x86_64	0.215	**4.39**×
RISC-V 64	0.326	6.65×
MIPS	0.446	9.10×
ARM64	0.472	9.63×
LiSa (x86)	1.437	29.3×
LiSa (ARM)	109.5	2235×
ElFEN	7.522	154×
QEMU VM (ARM)	1.109	22.6×
VM + `strace`	297.13	6063×

(b) Lifecycle Latency Comparison

Arch	Ours	LiSa [7]	Gain
Startup (ms)			
x86	509.9	5413	**10.6**×
MIPS	511.8	7487	**14.6**×
ARM	597.3	2900	**4.8**×
RISC-V	567.2	N/A	–
Teardown (ms)			
x86	104.2	5551	**53.3**×
MIPS	99.2	3336	**33.6**×
ARM	106.3	3311	**31.1**×
RISC-V	97.0	N/A	–

Startup: from creation to probe readiness.

Teardown: from execution completion to shutdown.

End-to-End Performance and Cost. We evaluated end-to-end throughput on (N=176) samples. *Local Mode* processing (without external API calls) took 108.82 s. Enabling the DeepSeek API (concurrency = 7) increased the total time to 438.94 s, with the added latency dominated by remote inference and throttling effects rather than local tracing. CPU utilization indicates effective parallel execution, averaging 69.92% and peaking at 308.85% (aggregated across CPU cores). Kernel-mode CPU utilization averaged 24.28%, consistent with container scheduling and frequent process creation overhead. Memory usage remained stable between 9.41–10.00 GiB, without sustained upward trends, suggesting no progressive memory growth under our cgroup limits per container. Based on DeepSeek's token pricing($0.286 per million uncached input tokens, and $0.429 per million output tokens), analyzing 100 samples under our settings costs about $0.29.

4.5 Behavior Profile Extraction

To examine the information captured beyond binary verdicts, we extract a structured behavior profile from each LLM-generated report. We represent each sample as a binary vector $\mathbf{b} \in \{0,1\}^8$ across eight categories: Drop+Exec, Daemon, C2, Persistence, Recon, Privilege Impact, Exec. Failure, and Crypto/SSH/TLS. Each flag is determined by a hybrid rule combining keyword matching on specific report fields (`summary`, `suspicious_behaviors`, `key_evidence`) and direct system call trace evidence. Examples include matching CLONE and DUP2 for Daemon, SOCKET and CONNECT for C2, and KILL operations targeting PID 1 for Privilege

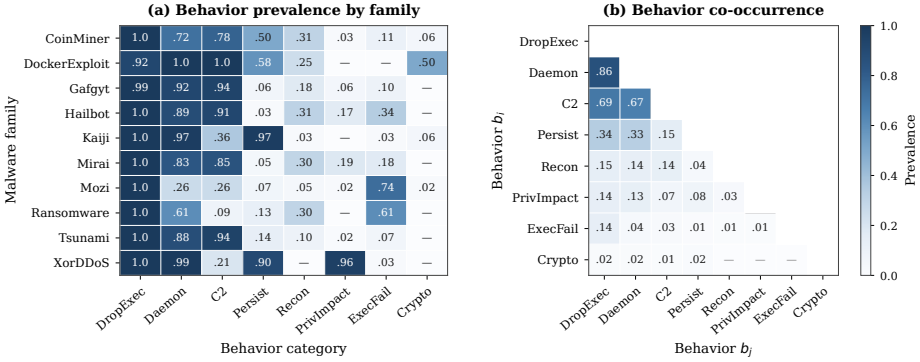

Fig. 6. Behavior profile over N=844 Linux malware samples. (a) Per-family prevalence $p_{f,k}$: the fraction of samples in each family exhibiting the behavior. (b) Global pairwise co-occurrence $p_{j,k}$ (lower triangle, where "—" indicates no co-occurrence).

Impact. We calculate the per-family prevalence $p_{f,k} = |F_f|^{-1}\sum_{i\in F_f} b_{i,k}$ and the global pairwise co-occurrence $p_{j,k} = N^{-1}\sum_i b_{i,j}b_{i,k}$ to analyze family-level patterns and dependencies between different behaviors.

Figure 6 displays profile distributions across the N=844 dataset samples. Drop+Exec appears in almost all samples ($p\approx$1.0 per family), indicating a common Linux malware deployment mechanism. Other categories show family-specific variations. Daemon and C2 behaviors are frequent in IoT botnets (e.g.,Gafgyt, Tsunami). Kaiji frequently shows Persistence (p=0.97), XorDDoS often exhibits Privilege Impact (p=0.96), and DockerExploit alone shows frequent Crypto/SSH/TLS activity. These differences confirm that the generated reports capture family-specific signals rather than applying a uniform classification.

The co-occurrence matrix in Figure 6(b) shows that Drop+Exec, Daemon, and C2 frequently appear together in most samples. Other combinations of behaviors appear in under 20% of the samples, varying by the malware family. This distribution suggests a common set of base operations among the analyzed Linux malware, alongside family-specific capabilities.

5 Case Study

We present a case study on a Mirai variant (`bd7a...da81`[2]) to examine analysis reliability under unstable execution. Table 10 contrasts MirrorShield with two commercial sandboxes. *Joe Sandbox* triggers a runtime panic (`sigaction failed`), aborts the run, and falls back to a verdict dominated by static YARA matches.[3] *Dr. Web vxCube* executes successfully but reports only coarse behavioral tags, lacking concrete targets or the surrounding context [48, 49]. Conversely, MirrorShield reconstructs an end-to-end attack narrative from runtime evidence, even under noisy or partially truncated execution. From the syscall trace,

[2]Full SHA-256: `bd7a37a86a1fa6831f426d1570f702aaffe454b4609eda5f111b7bd0f4d6da81`

[3]Report: sandbox report.

the LLM identifies the "Killer" routine by enumerating competitor artifacts removed via `UNLINKAT`, recovering the persistence mechanism by extracting the embedded shell loop and cron-based scheduling. This translates generic alerts into concrete findings: linking low-level events (files, scripts, process signals) to semantic roles (competitor removal, persistence) enables downstream triage and incident response.

Table 10. Comparative Analysis: Mirai Variant (Sample `bd7a...`)

Commercial (Joe / Dr.Web) *Status: Crash/Success*	**MirrorShield (Ours)** *Status: Success*
Joe Sandbox [19]: CRASHED. Verdict: Generic Malicious. Fallback: Static Yara Detection. **Dr. Web [48]:** Generic tags only (Deletes file, Runs as daemon). No targets.	**Forensic Reconstruction:** • **Killer:** `UNLINKAT` on `/etc/.ares`. • **Persistence:** Extracted shell loop & cron. • **Anti-Forensics:** Removed `libdlrpcld.so`. • **Flow:** Traced `CLONE` & `SIG23`.
Key Syscall Evidence	
`UNLINKAT("/etc/rc.d/init.d/linux_kill")`	→ *Competitor Removal*
`UNLINKAT("/usr/lib/libdlrpcld.so")`	→ *Monitor Deletion*
`WRITE(9,"#!/bin/sh\nwhile...")`	→ *Persistence: Script*
`READ(3, "/etc/init.d/sshd")`	→ *Recon: Service Enum*

6 Discussion and Future Work

Safety Trade-offs and Isolation Boundaries. While containerization offers the scalability central to MirrorShield, it provides weaker isolation than hardware virtualization. Containers share the host kernel, increasing the attack surface for privilege escalation. Container escape vulnerabilities, such as CVE-2019-5736 [50], CVE-2024-21626 [51], CVE-2022-0492 [52], and CVE-2022-0847 [53], remain critical concerns. To mitigate these risks, MirrorShield enforces *User Namespaces (UserNS)*. Empirical studies [54] confirm that UserNS neutralizes many runtime exploits, such as CVE-2019-14271 [55], by remapping the container's root to an unprivileged host user. For stronger isolation, containers could be wrapped in MicroVMs, confining kernel escapes while preserving deployment flexibility.

Limits of Syscall-Centric Analysis. MirrorShield captures system calls via eBPF to monitor OS interactions. However, this approach faces a *Semantic Gap*. First, purely user space activities (e.g., encryption loops or string de-obfuscation) are invisible until an I/O operation occurs. Second, advanced malware may use *Syscall Evasion* or shadow attacks [56] to split malicious sequences across processes. Future work could integrate uprobes or CPU performance counters to detect computation-heavy logic.

Transparent Artifacts and Evasion Risks. We acknowledge that `qemu-user` introduces detectable artifacts. However, studies show IoT malware like Mirai

prioritizes rapid infection over sophisticated evasion [57]. MirrorShield is thus optimized for high-throughput triage rather than absolute stealth; highly evasive samples remain better suited for resource-intensive bare-metal analysis [11]. Future work will scrub emulation fingerprints to reduce the detection surface.

7 Conclusion

We presented MirrorShield, a lightweight framework for cross-architecture Linux malware analysis. By combining containerized QEMU user-mode emulation with eBPF-based kernel monitoring, our system balances scalable execution with evidence-grounded interpretation. Budgeted sampling and local alerts prioritize high-signal behaviors, linking reports to observed runtime artifacts.

Our evaluation shows strong accuracy across diverse ISAs, including wild long-tail architectures, and robustness under extreme trace flooding. Testing four LLM backends yield 91.6% to 94.7% accuracy under identical evidence and prompts, confirming detection quality is driven by structured evidence rather than model choice. An *Evidence Match* evaluation shows that 81.4% to 95.0% of key claims trace to concrete execution artifacts across all backends. Behavior profile extraction reveals a stable $\langle$*Drop+Exec*, *Daemon*, *C2*$\rangle$ core triad shared across Linux malware families, alongside family-discriminative signatures for finer threat characterization.

Although container-based execution offers weaker isolation than virtualization, MirrorShield provides a practical, multiplexed environment eliminating per-architecture system-level setups.

Acknowledgments. We would like to thank the anonymous reviewers and COMPASS members for their insightful comments. This work was supported by the Special Funds (Grant No. pdjh2026bk186) for the Cultivation of Guangdong College Students' Scientific and Technological Innovation, and the National Natural Science Foundation of China under Grant No. U2541211, No. 62372218, and No. U24A6009. Finally, we acknowledge the CS315 Computer Security course at SUSTech, which provided the essential platform to develop the full system and complete this paper. We extend our sincere gratitude to Professor Hugh Anderson and the "Defence Against the Dark Arts" summer workshop at the National University of Singapore, where the initial concept and the prototype were conceived.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

Use of Generative AI. The authors used generative AI tools (Gemini and ChatGPT) to polish the language of the manuscript and to assist with code implementation. All research design, technical content, analyses, conclusions, and final code were authored, reviewed, and verified by the authors.

Appendix A Full LLM Analysis Report

LLM Analysis: Sample `bd7a...da81`

Final Prediction: **MALWARE** (95/100)

Family: `Mirai` **Status:** `Success`
Label: `Trojan.Linux.Mirai`
Categories: `Botnet, DDoS, Evasion`

Analysis Summary. This specimen is identified as a Mirai botnet variant featuring a "Killer" module. It aggressively scans for and removes competing malware (e.g., cleaning `/etc/.ares`), establishes persistence via shell scripts, and attempts to manipulate system cron jobs.

Key Findings (Extracted Artifacts):

- **Killer Behavior:** `UNLINKAT` events targeting competitor files: `/etc/.ares`, `/usr/lib/libdlrpcld.so`, `/etc/init.d/linux_kill`.
- **Persistence:** `WRITE` operations creating a loop script: `"#!/bin/sh\nwhile [ 1 ]; do\nsleep 60..."`.
- **System Tampering:** `EXEC` calls attempting to overwrite cron jobs (`*/1 * * * * root /.mod`).
- **Process Control:** Multiple `CLONE` and `SIGNAL_GENERATE` (sig 23) events.

Fig. 7. Structured report generated by the LLM for sample `bd7a...da81`, highlighting critical IOCs (e.g., "Killer" module targets) that traditional sandboxes missed due to runtime crashes.

Appendix B LLM Prompt Templates

System Prompt Specification

ROLE: Malware Analysis Expert.
TASK: Risk assessment from JSON data.

SECTION 1: ENVIRONMENT (CRITICAL)
- **ENV:** Docker(Linux), Binaries from `/tmp`, Non-x86 via QEMU.
- **NOISE FILTERING (BENIGN):**
  - Startup scripts in `/tmp`.
  - `TRACK_QEMU` events (Emulation artifact, NOT evasion).
  - IGNORE unless touching sensitive paths outside emulation.

SECTION 2: INPUT DATA
- **CONTEXT:** {exec_context}
- **DATA:**
  - `local_alerts`: Hints (verify with trace).
  - `syscall_trace`: Runtime events (Max 800).
  - `loop_patterns`: `_repeated_times` / `repeated_block`.

```
{summary_json}
```

SECTION 3: ANALYSIS LOGIC
- **STEP 1: STARTUP CHECK**
  IF fail (missing libs, exit!=0, no activity): STATUS=Failed_Dependencies, SCORE=0, STOP.
- **STEP 2: ENV COMPATIBILITY**
  IF early exit due to missing conditions (but benign): VERDICT=Unknown, SCORE=0.
- **STEP 3: EVASION DETECTION**
  Active anti-analysis only. NOTE: `TRACK_QEMU` != Evasion.
- **STEP 4: BEHAVIOR RECONSTRUCTION (CRITICAL)**
  Method: Reconstruct intent via `syscall_trace` TEMPORAL SEQUENCE.
  - *Dropper:* socket→connect→recv→open→write→execve.
  - *DDoS:* High vol `sendto`(UDP)/`connect`(TCP) in loops.
  - *Persistence:* open/write to `/etc/rc.local`, `crontab`, etc.
- **STEP 5: CLASSIFICATION**
  MALICIOUS (80-100), SUSPICIOUS (50-79), etc.

SECTION 4: FAMILY KNOWLEDGE BASE (Keywords)
- **MIRAI:** IoT DDoS. TCP C2(23, 2323). Procs(kakashi, anime). Exploit(Huawei).
- **GAFGYT:** Mirai-code-reuse. C2(IRC). Cmds(!udp, !tcp).
- **KINSING:** Miner worm. Procs(kdevtmpfsi). Kills XMRig rivals.
- *... (Truncated for brevity if needed) ...*

SECTION 5: OUTPUT FORMAT (Pure JSON)

```
{{
    "risk_score": <0-100>,
    "verdict": "<Malicious|Benign...>",
    "threat_categories": ["<Cat1>", "<Cat2>"],
    "family": "<Name|CVE|None>",
    "analysis": {{
        "summary": "...",
        "key_evidence": ["..."]
    }}
}}
```

References

1 Antiy Labs. *Annual Growth Trend of Effective Malware Samples*. https://www.virusview.net/statistics. Accessed: 2025-12-07 (in Chinese). 2025.

2 Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. "Understanding Linux Malware". In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2018, pp. 161–175.

3 Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. "PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware". In: *22nd Annual Computer Security Applications Conference (ACSAC)*. IEEE, 2006, pp. 289–300. DOI: 10.1109/ACSAC.2006.38.

4 Claudio Guarnieri, Alessandro Tanasi, Jurriaan Bremer, and Michael Schloesser. "The Cuckoo Sandbox". In: *Black Hat USA*. Open Source Software. 2012. URL: https://cuckoosandbox.org/.

5 K. A. Monnappa. *Limon: Sandbox for Analyzing Linux Malware*. https://github.com/monnappa22/Limon. Accessed: 2025-12-07. 2015.

6 Detux Team. *Detux: Multiplatform Linux Sandbox*. https://github.com/detuxsandbox/detux. 2015.

7 Daniel Uhríček. *LiSa – Multiplatform Linux Sandbox for Analyzing IoT Malware*. https://excel.fit.vutbr.cz/submissions/2019/058/58.pdf. 2019.

8 ANY.RUN. *ANY.RUN API documentation*. https://any.run/api-documentation/. Accessed: 2026-01-31.

9 Nikhil Hegde. *ELFEN: Automated Linux Malware Analysis Sandbox*. Presentation at Nullcon Goa 2023. Slides available at GitHub; Talk available at https://www.youtube.com/watch?v=opfwbNlijSg. Accessed: 2026-01-31. 2023. URL: https://github.com/nikhilh-20/ELFEN.

10 Xu Chen, Jon Andersen, Z. Morley Mao, Michael Bailey, and Jose Nazario. "Towards an Understanding of Anti-virtualization and Anti-debugging Behavior in Modern Malware". In: *Dependable Systems and Networks (DSN)*. IEEE, 2008, pp. 177–186. DOI: 10.1109/DSN.2008.4630086.

11 Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. "BareCloud: Bare-metal analysis-based evasive malware detection". In: *23rd USENIX Security Symposium (USENIX Security 14)*. 2014, pp. 287–301.

12 Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. "Repeatable reverse engineering with PANDA". In: *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*. 2015, pp. 1–11.

13 Zhenyu Ning and Fengwei Zhang. "Ninja: Towards Transparent Tracing and Debugging on ARM". In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 33–49. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/ning.

14 Fengwei Zhang, Kevin Leach, Angelos Stavrou, and Haining Wang. "Towards Transparent Debugging". In: *IEEE Transactions on Dependable and Secure Computing* 15.2 (2018), pp. 321–335. DOI: 10.1109/TDSC.2016.2545671.

15 Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. "Lost in the Middle: How Language Models Use Long Contexts". In: *Transactions of the Association for Computational Linguistics* 12 (2024), pp. 157–173. DOI: 10.1162/tacl_a_00638.

16 Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2020.

17 Ilia Shumailov, Yiren Zhao, Daniel Bates, Nicolas Papernot, Robert Mullins, and Ross Anderson. "Sponge Examples: Energy-Latency Attacks on Neural Networks". In: *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2021, pp. 212–231.

18 Mohamed Amine Ferrag, Norbert Tihanyi, Djallel Hamouda, Leandros Maglaras, Abderrahmane Lakas, and Merouane Debbah. "From prompt injections to protocol exploits: Threats in LLM-powered AI agents workflows". In: *ICT Express* (2025). In press. ISSN: 2405-9595. DOI: 10.1016/j.icte.2025.12.001. URL: https://www.sciencedirect.com/science/article/pii/S2405959525001997.

19 Joe Security AG. *Joe Sandbox: Advanced Malware Analysis Platform*. https://www.joesecurity.org/. 2023.

20 Tamas K Lengyel, Steve Maresca, Bryan D Payne, George D Webster, Sebastian Vogl, and Aggelos Kiayias. "Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system". In: *Proceedings of the 30th Annual Computer Security Applications Conference*. 2014, pp. 386–395.

21 Cisco Talos. *PyREBox: Python Scriptable Reverse Engineering Sandbox*. https://github.com/Cisco-Talos/pyrebox. Accessed: 2026-02-14. 2017.

22 VMRay GmbH. *VMRay Analyzer: Agentless Hypervisor-Based Malware Analysis Sandbox*. https://www.vmray.com/vmray-analyzer/. Accessed: 2026-01-25. 2024. URL: https://www.vmray.com/vmray-analyzer/.

23 Google LLC. *gVisor: Container Runtime Sandbox*. https://gvisor.dev/. 2023.

24 OpenInfra Foundation. *Kata Containers: Secure Container Runtime*. https://katacontainers.io/. 2023.

25 Amazon Web Services. *Firecracker: Secure and Fast MicroVMs*. https://firecracker-microvm.github.io/. 2018.

26 Sysdig Inc. *Falco: Cloud-Native Runtime Security with eBPF*. https://falco.org/. 2023.

27 Hyrum S Anderson and Phil Roth. "EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models". In: *arXiv preprint arXiv:1804.04637*. 2018.

28 Xin Xiao, Shutao Zhang, Francesco Mercaldo, Guangwu Hu, and Arun Kumar Sangaiah. "Android Malware Detection Based on System Call Sequences and LSTM". In: *Multimedia Tools and Applications* 78 (2019), pp. 3979–3999.

29 Liting Deng, Hui Wen, Mingfeng Xin, Hong Li, Zhiwen Pan, and Limin Sun. "Enimanal: Augmented cross-architecture IoT malware analysis using graph neural networks". In: *Computers & Security* 132 (Sept. 2023), p. 103323. DOI: 10.1016/j.cose.2023.103323.

30 Senming Yan, Jing Ren, Wei Wang, Limin Sun, Wei Zhang, and Quan Yu. "A Survey of Adversarial Attack and Defense Methods for Malware Classification in Cyber Security". In: *IEEE Communications Surveys & Tutorials* 25.1 (2023), pp. 467–496. DOI: 10.1109/COMST.2022.3225137.

31 In3tinct. *Androidmeda: LLM tool for deobfuscating Android apps and finding vulnerabilities*. GitHub repository. https://github.com/In3tinct/Androidmeda. 2024.

32 Reza Fayyazi, Rozhina Taghdimi, and Shanchieh Jay Yang. "Advancing TTP Analysis: Harnessing the Power of Large Language Models with Retrieval Augmented Generation". In: *2024 Annual Computer Security Applications Conference Workshops (ACSAC Workshops)*. 2024, pp. 255–261. DOI: 10.1109/ACSACW65225.2024.00036.

33 Yiling He, Hongyu She, Xingzhi Qian, Xinran Zheng, Zhuo Chen, Zhan Qin, and Lorenzo Cavallaro. "On benchmarking code llms for android malware analysis". In: *Proceedings of the 34th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2025, pp. 153–160.

34 Chongzhou Fang, Ning Miao, Shaurya Srivastav, Jialin Liu, Ruoyu Zhang, Ruijie Fang, Asmita, Ryan Tsang, Najmeh Nazari, Han Wang, and Houman Homayoun. "Large language models for code analysis: Do LLMs really do their job?" In: *33rd USENIX Security Symposium (USENIX Security 24)*. 2024, pp. 829–846.

35 Coleman Richard Charles Hooper, Sehoon Kim, Hiva Mohammadzadeh, Monishwaran Maheswaran, Sebastian Zhao, June Paik, Michael W Mahoney, Kurt Keutzer, and Amir Gholami. "Squeezed attention: Accelerating long context length llm inference". In: *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2025, pp. 32631–32652.

36 *ptrace(2) — process trace*. Linux man-pages project. Manual page, Accessed: 2026-02-08. URL: https://man7.org/linux/man-pages/man2/ptrace.2.html.

37 Michael Kerrisk. *namespaces(7) - Linux manual page*. https://man7.org/linux/man-pages/man7/namespaces.7.html. 2024.

38 Sysdig Inc. *2023 Cloud-Native Security and Usage Report*. https://www.scribd.com/document/639835552/2023-cloud-native-security-and-usage-report. Downloaded from Scribd. Original report published by Sysdig, Inc., covers cloud-native security and usage trends. 2023.

39 Murugiah Souppaya, John Morello, and Karen Scarfone. *Application Container Security Guide (NIST Special Publication 800-190)*. Tech. rep. https://doi.org/10.6028/NIST.SP.800-190. National Institute of Standards and Technology, 2017.

40 Linux Kernel Organization. *Kernel Support for miscellaneous Binary Formats (binfmt_misc)*. https://www.kernel.org/doc/html/latest/admin-guide/binfmt-misc.html. 2024.

41 The QEMU Project Developers. *QEMU User Space Emulation Documentation*. Accessed: 2025-11-25. 2024.

42 Abuse.ch. *MalwareBazaar: A value-free malware exchange*. Accessed: 2025-12-14. 2025. URL: https://bazaar.abuse.ch/.

43 VirusTotal. *VirusTotal - Free Online Virus, Malware and URL Scanner*. Accessed: 2025-12-14. 2025. URL: https://www.virustotal.com/.

44 DeepSeek. *Models & Pricing (DeepSeek API Docs): DeepSeek-V3.2*. https://api-docs.deepseek.com/quick_start/pricing. Accessed: 2026-02-07. 2025.

45 Hatching International B.V. *Hatching Triage*. Accessed: 2026-01-31. URL: https://hatching.io/triage/.

46 Larry McVoy and Carl Staelin. "lmbench: Portable Tools for Performance Analysis". In: *USENIX Annual Technical Conference*. 1996, pp. 279–294.

47 Hai-Viet Le and Quoc-Dung Ngo. "V-Sandbox for Dynamic Analysis IoT Botnet". In: *IEEE Access* 8 (2020), pp. 145768–145786.

48 Doctor Web. *Dr. Web vxCube*. Accessed: 2025-12-14. 2025. URL: https://vxcube.drweb.com/.

49 Dr. Web. *vxCube Analysis: Mirai Variant*. Accessed: 2026-01-27. 2025. URL: https://bazaar.abuse.ch/sample/bd7a37a86a1fa6831f426d1570f702aaffe454b4609eda5f111b7bd0f4d6da81/.

50 NIST National Vulnerability Database. *CVE-2019-5736: runc Container Escape Vulnerability*. https://nvd.nist.gov/vuln/detail/CVE-2019-5736. 2019.

51 NIST National Vulnerability Database. *CVE-2024-21626: runc File Descriptor Leak Vulnerability*. https://nvd.nist.gov/vuln/detail/CVE-2024-21626. 2024.

52 NIST National Vulnerability Database. *CVE-2022-0492: Linux Kernel cgroups release_agent Privilege Escalation*. https://nvd.nist.gov/vuln/detail/CVE-2022-0492. 2022.

53 NIST National Vulnerability Database. *CVE-2022-0847: Linux Kernel Privilege Escalation via Dirty Pipe*. https://nvd.nist.gov/vuln/detail/CVE-2022-0847. 2022.

54 Michael Reeves, Dave Jing Tian, Antonio Bianchi, and Z Berkay Celik. "Towards improving container security by preventing runtime escapes". In: *2021 IEEE Secure Development Conference (SecDev)*. IEEE. 2021, pp. 38–46.

55 NIST National Vulnerability Database. *CVE-2019-14271: Docker cp Code Injection Vulnerability*. https://nvd.nist.gov/vuln/detail/CVE-2019-14271. 2019.

56 Weiqin Ma, Pu Duan, Sanmin Liu, Guofei Gu, and Jyh-Charn Liu. "Shadow attacks: automatically evading system-call-behavior based malware detection". In: *Journal in Computer Virology* 8.1-2 (2012), pp. 1–13. DOI: 10.1007/s11416-011-0157-5.

57 Benjamin Vignau, Raphaël Khoury, Sylvain Hallé, and Abdelwahab Hamou-Lhadj. "The evolution of IoT Malwares, from 2008 to 2019: Survey, taxonomy, process simulator and perspectives". In: *Journal of Systems Architecture* 116 (2021), p. 102143.