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

No Author Given

No Institute Given

Abstract. Existing Linux malware analysis tools often struggle to scale across diverse CPU architectures, while many automated detectors provide only coarse-grained verdicts with limited evidence for forensic analysis. 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 through an eBPF-based kernel monitor. To make LLM-based reporting reliable under tight context budgets, MirrorShield curates traces using local analyzers together with denoising, sampling, and structured prompting, transforming low-level logs into evidence-backed reports. Across 11 CPU architectures, MirrorShield achieves 94.7% detection accuracy (F1: 0.946). MirrorShield remains robust under extreme context flooding, maintaining over 90% detection even with 2,000× noise injection. We further evaluate report reliability using an Evidence Match metric and find that 80.7% of key report claims are directly supported by observed trace artifacts. In addition, MirrorShield reduces analysis turnaround time and overhead compared to common baselines, enabling scalable end-to-end analysis in practice.

Keywords: Malware Analysis · Dynamic Analysis · Cross-Architecture · Internet of Things (IoT) · Large Language Models.

1 Introduction

The past decade has witnessed a sustained surge in malware volume and diversity, [1] while historically focused on Windows, the threat landscape has shifted significantly toward Linux, driven by the proliferation of IoT and embedded systems. A recent example is the DDoS attack targeting DeepSeek in early 2025, orchestrated by Linux-based families like HailBot and RapperBot [2]. Analyzing Linux malware, however, presents distinct challenges compared to Windows, primarily due to its complex binary compatibility [3]. Unlike the standardized Windows environment, Linux malware targets diverse CPU architectures (e.g., x86, MIPS, ARM) and relies on specific runtime dependencies, such as custom loaders and varying system libraries. Although static analysis is attractive for being architecture-agnostic, it is frequently undermined by statically linked binaries (over 80%) and aggressive packing/obfuscation that hides semantics and injects substantial noise [3, 4]. Consequently, defenders turn to dynamic analysis to recover runtime intent.

Dynamic analysis can reveal concrete behaviors (e.g., filesystem, process, network, and persistence) that are 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 at scale [5, 6, 7, 8, 9, 10], which exposes a fundamental tension between scalability and transparency. Modern malware increasingly fingerprints virtualized or monitored environments and may suppress malicious actions, motivating research on transparent observation and evasion-resistant analysis [11, 12]. To mitigate virtualization artifacts, many systems move deeper into the stack via virtual machine introspection (VMI) and record/replay-style instrumentation (e.g., DRAKVUF and PANDA) [13, 14]. Others pursue hardware-assisted or bare-metal analysis that is harder to detect, such as Ninja or MALT [15, 16]. While these methods improve transparency, they incur high overhead and deployment complexity. Consequently, they are better suited for deep single-sample analysis. This motivates a lightweight and easy-to-deploy analysis framework that can scale to large sample volumes while still capturing behavior that is meaningful for analysis.

At the same time, automated detection often trades interpretability for accuracy: rule-based signatures are fragile under packing and obfuscation, while black-box ML models provide limited forensic justification. LLMs can bridge this gap by translating low-level execution traces into human-readable intent and actionable IOCs. However, producing reliable forensic narratives from long, low-level traces requires careful context construction: the evidence is often sparse and scattered, while prompt budgets are limited. Without structured, evidence-centered inputs, LLMs may miss critical signals in long contexts [17] or introduce unsupported details, motivating explicit grounding and evidence-aware summarization [18].

We introduce MirrorShield, an evidence-grounded framework for cross-architecture Linux malware analysis that scales by pairing lightweight emulation with kernel-level telemetry and LLM reporting. To fit within tight LLM budgets without adding heavy overhead, 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 that MirrorShield achieves 94.7% detection accuracy (F1: 0.946) while generalizing well across architectures. In terms of efficiency, the pipeline incurs minimal runtime overhead ($< 10\times$), and is substantially faster than full-system emulation in our setting, where slowdowns can exceed 20,000×. To assess robustness under limited context budgets and noisy traces, we conduct extreme context flooding and find that detection remains above 90% even with up to 2,000× noise injection [19, 20], with the added cost saturating as noise increases. Finally, to quantify report reliability, we introduce an *Evidence Match* evaluation: across 844 reports, 80.7% of key evidence claims can be directly linked to observable trace artifacts (3.86 supported items per report on average), providing trace-level support for the generated conclusions. Our contributions are:

– We build MirrorShield on containerized QEMU user-mode execution and eBPF-based syscall tracing, supporting 11 CPU architectures for scalable,

end-to-end malware analysis on commodity x86 hosts. We will open-source MirrorShield at an anonymized repository and release our evaluation data.
- We propose a novel curation pipeline that denoises, samples, and prioritizes alert-relevant events to fit execution evidence into limited LLM context windows.
- Beyond detection metrics, we assess report quality by measuring conclusion correctness and whether key claims can be verified from execution traces.
- We validate MirrorShield on a diverse cross-architecture dataset, achieving 94.7% detection accuracy (F1: 0.946) while reducing overhead and analysis turnaround time compared to traditional Linux sandboxes.

2 Related Work and Scope

2.1 Dynamic Malware Analysis Platforms

Agent-Hook-Based Sandboxes. Early platforms relied on in-guest agents to intercept system calls and APIs. For instance, Cuckoo Sandbox employs API hooking via in-guest components to monitor behavior [5], whereas LIMON serves as a wrapper around `strace` for tracing execution paths [6]. Other works for IoT threats, such as Detux and LISA, focused on capturing network traffic or provided multi-architecture sandboxes [7, 8]. Commercial solutions like Joe Sandbox also offer cloud-based cross-platform capabilities [21]. Nevertheless, their monitors reside within the same privilege domain as the malware, introducing detectable artifacts and making them vulnerable to evasion techniques.

VMI and Hardware-Assisted Approaches. To address evasion, a subsequent line of research moved monitoring outside the guest OS using Virtual Machine Introspection (VMI). DRAKVUF,for example, leverages Xen and LibVMI to transparently track system calls for rootkit analysis [13]. PANDA provides record-and-replay capabilities with deep taint analysis [14] and PyREBox enables scriptable instrumentation of running systems [22]. Similarly, commercial solutions like VMRay Analyzer use hypervisor-level monitoring to remain stealthy [23]. Hardware-assisted frameworks, such as Ninja and MalT, utilize ARM TrustZone or System Management Mode to further hide the monitor [15, 16]. Despite their improved stealth, the record-and-replay mechanism enabled by PANDA causes a 4× slowdown, and heavyweight analysis plugins (e.g., taint tracking) can increase overhead by up to 100× [14], rendering them difficult to scale for high-throughput analysis.

Container-Based Isolation. Modern secure runtimes, like gVisor, aim to achieve lightweight isolation, but their use of user-space syscall interception could potentially conceal actual malware behaviors. Other solutions like micro-VMs (e.g., Kata, Firecracker) reintroduce resource redundancy when deploying monitoring agents [24, 25, 26]. Furthermore, existing eBPF tools like Falco are optimized for runtime defense rather than the granular forensic introspection [27].

2.2 Automated Detection and Interpretation

Machine Learning-Based Detection. Machine Learning (ML) approaches automate the classification of malware based on extracted features. Works like EMBER utilize static features from PE files or raw bytes to train classifiers [28]. For dynamic behaviors, Xiao et al. applied LSTM networks to model system call sequences in Android malware [29]. Deng et al. proposed ENIMANAL for IoT malware, , which converts system call traces into an attributed system call graph and applies GNN-based classification, enhancing robustness via semantic augmentation from system call specifications [30]. Despite their effectiveness, these black-box models offer limited explainability and show poor generalization to zero-day exploits and adversarial examples [31].

LLM-Assisted Analysis. LLMs show great potential in semantic malware analysis, including deobfuscation (e.g., Androidmeda [32]) and TTP generation [33], as further evidenced by benchmarking frameworks for Code LLMs on Android malware [34]. However, their use in dynamic analysis is limited. Research has revealed that LLMs frequently suffer from hallucinations and shallow reasoning when analyzing complex code [35]. while finite context windows and high inference costs make processing voluminous raw execution traces impractical [36]. In contrast to these works, our research addresses this gap by bridging lightweight kernel-level monitoring with LLMs while effectively managing the 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 that flood traces with noise or exploit limited LLM context budgets [19, 20]. To reduce reliance on user-space tracing that is easier to detect and disrupt [37, 11, 12], 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 or emulation escape, host-kernel exploits, and kernel rootkits that tamper with telemetry.

3 System Design and Implementation

Figure 1 illustrates the high-level architecture of MirrorShield. The analysis workflow begins in the isolation environment (Section 3.1). Here, the controller runs cross-architecture binaries concurrently using Docker containers and QEMU-USER emulation. Simultaneously, the kernel space layer (Section 3.2) captures runtime behaviors using eBPF, employing Cgroup-based filtering to ensure high-speed and noise-free data collection. Finally, the captured event stream is routed to the corresponding sample and subsequently triggers the analysis engine (Section 3.3). First, a custom-developed local analyzer performs rapid behavioral profiling. If the system call stream exceeds the LLM context limit, it applies sampling to reduce redundancy. Finally, the retained signals are combined with container execution output, formatted into task-specific prompts, and sent to an LLM for high-level intent reconstruction.

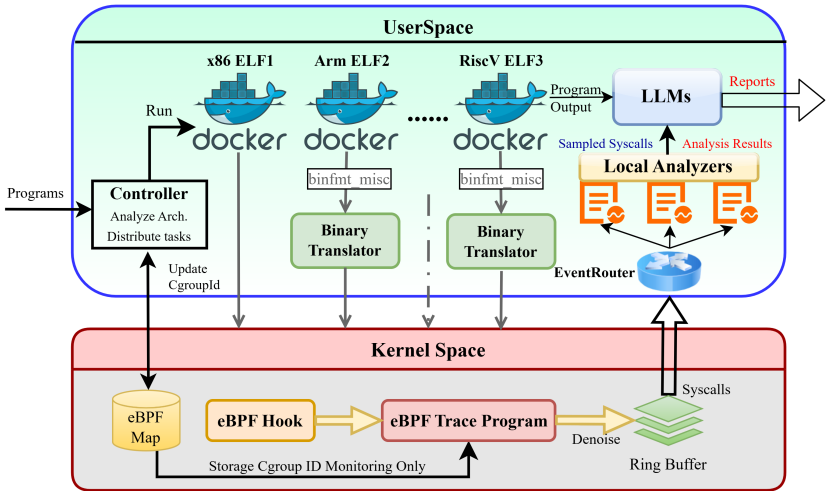

Fig. 1. System Design Diagram

3.1 Execution Workflow and Environment

The execution process begins with MirrorShield traversing the target directory. It parses the ELF header of each binary to identify its Instruction Set Architecture (ISA) and library dependencies, matching them with the appropriate Docker image which is selected from our curated collection of official and secure images. Subsequently, the controller concurrently spawns multiple Docker containers in a dormant state. During this initialization phase, MirrorShield extracts the Cgroup ID of each container and registers it into the BPF map. This mapping allows the kernel-level tracer to associate system events with specific containers. Once the environment is prepared, the target binary is injected via `docker cp` and executed using `docker exec`. Upon process termination or execution

timeout, the system enforces a cleanup routine, terminating the processes and removing the containers to release resources.

Isolation Configuration. To ensure host security, we configure the execution environment to leverage Docker's native isolation capabilities, as summarized in Table 1. Specifically, we enable the User Namespace feature [38], which uses *ID Mapping* to map the root user inside the container to an unprivileged user on the host system. In practice, industry reports show that over 83% of containers effectively execute with host-root privileges[39], largely due to the lack of user isolation in default runtimes. This configuration prevents potential privilege escalation attacks from affecting the host kernel, while simultaneously restricting its actual capabilities on the host kernel [40]. Additionally, we utilize Cgroups to enforce strict resource quotas, limiting CPU bandwidth and memory usage to prevent resource exhaustion during the analysis of potentially malicious binaries.

Table 1. 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

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 [41] 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 running a guest OS [42], 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 through the kernel's standard process launch path.

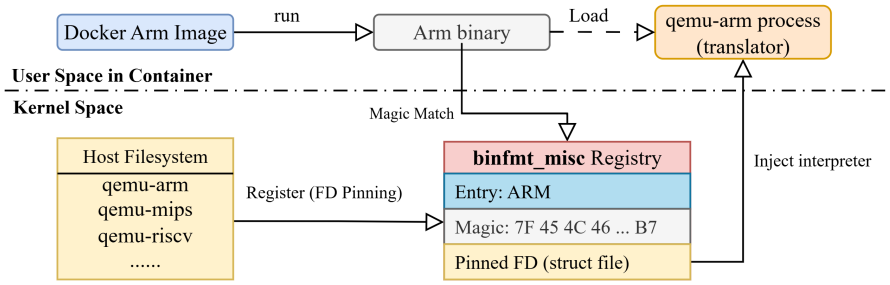

Fig. 2. FD-pinned binfmt interpreter for cross-architecture execution

3.2 Kernel-Space Observability

We implemented the core monitoring agent using eBPF. This architecture ensures kernel-level visibility and portability across different kernel versions without requiring recompilation.

To maintain high fidelity and resist user-space evasion, we operate entirely within the kernel using Tracepoints. Unlike `kprobes`, which hook internal kernel symbols subject to change, Tracepoints provide a stable ABI. We defined a targeted set of hooks in `monitor.bpf.c`, prioritizing security-critical behaviors over standard I/O operations. As exemplified in Table 2, our interception logic covers five critical behavioral dimensions, ensuring that we capture high-value adversarial behaviors such as fileless execution via `memfd_create` or debugger detection via `ptrace`—while ignoring irrelevant system noise. Beyond syscall identifiers, our probes extract selected arguments that define the semantic context of each call (e.g., file paths, flags, PIDs, and namespace or memory attributes).

Table 2. 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.

Dynamic Cgroup-based Filtering. Applying these hooks system-wide would introduce unacceptable performance overhead and lots of noise. To address this, we implemented a context-aware filtering mechanism that precisely targets analysis containers without requiring in-guest agents.

The filtering logic relies on synchronization between 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 show our main logic in Listing 1.1.Upon capturing a system call, the in-kernel eBPF probe applies filtering logic and caches the PID of the target process to optimize subsequent lookups.

Listing 1.1. Dynamic Cgroup filtering logic.

```
/* Maps: Active containers (Cgroup ID) and PID cache */
struct { __uint(type, BPF_MAP_TYPE_HASH); __type(key, u64); __type(value,
    u8); } monitored_cgroups SEC(".maps");
struct { __uint(type, BPF_MAP_TYPE_HASH); __type(key, u32); __type(value,
    u8); } monitored SEC(".maps");
static __always_inline bool should_monitor(u32 pid) {
    if (bpf_map_lookup_elem(&monitored, &pid)) return true;
    u64 cg = bpf_get_current_cgroup_id();
    if (bpf_map_lookup_elem(&monitored_cgroups, &cg)) {
        u8 val = 1; // Update PID cache
        bpf_map_update_elem(&monitored, &pid, &val, BPF_ANY);
        return true;
    }
    return false;
}
```

In-Kernel Noise Reduction. A major challenge in dynamic analysis is the high volume of benign system calls (e.g., standard library loading, memory allocations) that degrades performance. To mitigate this, we aggregate repetitive events directly within the kernel to filter redundancy at the source. We utilize two distinct aggregation strategies. First, for file system noise, we implemented 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 generating a log entry for every `mmap` syscall, we track the total execution count and the 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

To transform high-volume raw kernel events into actionable intelligence, we implemented a comprehensive analysis pipeline illustrated in Figure 3. The raw system calls are processed along two parallel paths: one stream is aggregated for Local analyzers to identify and alert on suspicious activities, while the other stream undergoes denoising before being fed directly into the LLM. The LLM acts as the central reasoning engine, integrating these local alerts and denoised syscalls—along with specific prompts and raw binary output—to generate a final, detailed report.

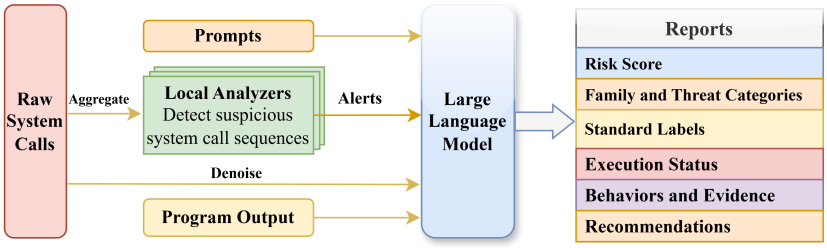

Fig. 3. AI-Assisted Two-Stage Behavioral Analysis Architecture

Data Aggregation and Rule-Based Detection. Raw system call logs are voluminous and noisy. To prepare this data, we first use a sliding-window algorithm to detect and compress repetitive sequences (e.g., tight `read/write` loops). This step significantly reduces data volume while preserving the core execution logic. The aggregated event stream is first processed by a suite of lightweight *local analyzers* before invoking any LLM. These analyzers are derived from common malicious behavior sequences observed in Linux malware (e.g., reconnaissance → privilege/escape attempts → persistence and C2), and we implement over 20 specialized modules to cover recurring tactics. An `AnalyzerManager` maintains per-sample in-memory state and routes events to the corresponding modules, enabling temporal correlation across syscalls and kernel events. For example, `KernelManipulation` tracks module-loading activity via `INIT_MODULE/FINIT_MODULE`, while `ContainerEscape` monitors namespace and filesystem pivot operations (e.g., `unshare`, `pivot_root`) to flag isolation breaches. In addition, analyzers perform argument- and policy-level matching against high-confidence patterns: `MemoryCorruption` detects suspicious `mprotect` transitions that introduce executable pages (`PROT_EXEC`), and `SandboxBypass` captures critical policy downgrades such as disabling `SECCOMP` via `prctl`.

Trace Denoising and Context Construction. Despite initial aggregation, the trace length may still exceed the context window of the LLM. To address this, we implement a priority sampling strategy before API 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 context for the LLM is constructed by synthesizing three key data sources:(i) the denoised syscall Trace, (ii) local analysis alerts, and (iii) the raw binary output. This comprehensive, synthesized context is then immediately wrapped into a structured prompt, serving as the definitive instruction set for the final reasoning stage.

LLM-Driven Reasoning. The fully structured prompt, enriched with deterministic local alerts and the denoised trace, is submitted to the LLM. We instruct the LLM to act as a `Malware Analysis Expert`, grounding its semantic

reasoning on local analyzer alerts. Under DeepSeek's limited context window, these alerts function as high-confidence reminders for potentially unseen syscalls. We also adopt a conservative alerting strategy so that analyzer outputs guide the model without imposing rigid rules. The prompt is carefully designed and includes lightweight domain priors (Appendix B). The model is then specifically tasked with three primary objectives. First, it must contextualize the analysis by correlating the rule-based alerts with the surrounding syscall trace and binary output to confirm validity. Second, it needs to reconstruct the high-level intent from the fragmented low-level events. Finally, the LLM must classify the threat, which involves assigning a final verdict and risk score (0 - 100) and outputting a standardized report that includes 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. Our evaluation is organized into two primary sections: accuracy verification and performance assessment. We evaluate MirrorShield end-to-end around four practical questions.

RQ1. How accurate is MirrorShield in malware/benign classification, and how much do the results change when we switch the underlying LLM (Sec. 4.1)?
RQ2. Compared with representative sandboxes that support limited Linux/ISA settings, how broad is MirrorShield's architecture coverage, and can it keep high accuracy across diverse ISAs and under noisy traces (Sec. 4.2)?
RQ3. How accurate and reliable are the report conclusions (Sec. 4.3)?
RQ4. Is the system lightweight enough for high-throughput analysis, and how much faster is it than common Linux sandbox baselines (Sec. 4.4)?

Dataset Construction. To ensure dataset heterogeneity, we collected 844 malware samples from MalwareBazaar [43]. We selected samples representing 9 distinct threat signatures based on the annotations of the database. Subsequently, we cross-referenced their `SHA-256` hashes with VirusTotal [44] to retrieve detailed heuristics about their malicious behaviors and family classifications. As illustrated in Fig. 4, our dataset encompasses nearly all prevalent CPU architectures and exhibits a wide range of malicious behaviors, predominately concentrating on botnet-related activities. 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]

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

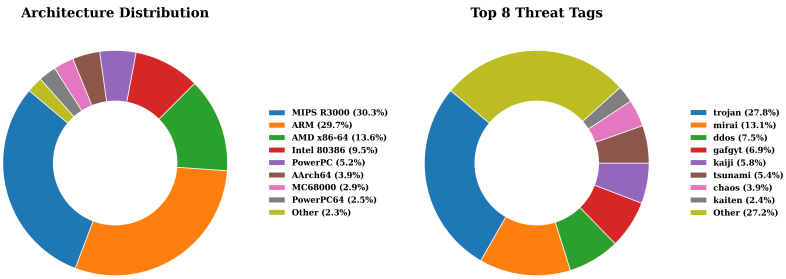

Fig. 4. Distribution of the collected malware dataset. The left chart depicts the top threat tags, while the right chart shows the targeted CPU architectures.

4.1 Detection Effectiveness

Overall Accuracy. Using DeepSeek V3.2 [45] as the default analyzer, as shown in Table 3(a), the pipeline achieves 94.7% accuracy **with an F1 score of 0.946**. Performance correlates with execution stability: accuracy reaches 96.17% on successfully executed samples but decreases to 90.95% for the 11.97% of cases involving crashes or dependency failures (Table 3(b)). While detection rates exceed 90% for most families, performance declines for *Ransomware* and *Hailbot*, largely due to dormant behaviors that require specific arguments or environmental triggers. Conversely, the 94.8% specificity on benign files indicates that false positives mainly involve administrative utilities such as *netcat* and *curl*, whose legitimate yet sensitive operations can resemble malicious actions. To assess analyzer sensitivity, we keep the same structured evidence and prompt but swap the LLM backend from DeepSeek V3.2 to Qwen Plus. [46] Accuracy decreases from 94.7% to 91.6% (F1=0.916) after swapping the analyzer suggesting that the pipeline remains effective across multiple LLM backends given the same structured evidence and prompt.

4.2 Cross-Architecture Generalization and Robustness

Architecture Coverage. Unlike existing sandboxes confined to x86 or specific ARM versions, our system unifies dynamic analysis across a broad spectrum of ISAs, demonstrating effective generalization beyond the architectures supported by prior work. The results show that mainstream architectures are robust, achieving average detection rates of over 93%. Table 4.2 summarizes our architecture support. Beyond mainstream ISAs, the pipeline also covers emerging targets such as 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 and still extract 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.

Table 3. Detailed Analysis: Detection Breakdown & Architecture Support

(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%**

Acc._{DS}: DeepSeek V3.2 Acc._{Q}: Qwen Plus

(b) Arch. Coverage & Stability

Arch	Count	Acc.	Exec. Err
x86 Family	563	94.5%	12.6%
ARM Family	695	92.4%	11.5%
MIPS	266	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%**	**12.0%**

Acc. is reported based on DeepSeek-V3.2 outputs.
Exec. Err indicates the program did not exit correctly (e.g., non-zero exit, crash, or timeout).

Table 4. Architecture support comparison.

Sandbox	x86	ARM	MIPS	RISC-V	PowerPC	s390x
Cuckoo [5]	✓	✓	–	–	–	–
Lisa [8]	✓	✓	✓	–	–	–
Detux [7]	✓	✓	✓	–	–	–
Limon [6]	✓	–	–	–	–	–
Triage [47]	✓	✓	✓	–	–	–
Any.Run [9]	✓	✓	–	–	–	–
ELFEN [10]	✓	32bit	✓	–	✓	–
Ours	✓†	✓†	✓†	✓	✓‡	✓

† 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 Stability. Complementing these results, Table 3(b) verifies robustness across heterogeneous architectures. Although mainstream architectures (x86, ARM) demonstrate stable execution, MIPS and RISC-V experience higher error rates close to 18%,which are primarily caused by the inherent instability of Linux malware. Specifically, many samples rely on hardcoded memory addresses for shellcode injection or specific dynamic libraries, which often trigger crashes in emulated environments. Moreover, MIPS samples often represent legacy IoT artifacts, their age introduces natural incompatibilities with modern runtime environments. Nevertheless, our system remains effective because the eBPF probe captures the critical syscall sequence preceding a crash. This allows the LLM to infer malicious intent from partial traces, which explains why detection accuracy is maintained despite execution instability.

Robustness to Context Flooding. To evaluate the system's resilience against context flooding attacks, we conducted a stress test on 50 randomly sampled malware instances. We developed an injection algorithm to interleave benign noise into the execution traces. For every malicious system call, we injected λ benign noise events immediately following it. To prevent trivial filtering, these injected events mimic the context of the preceding malicious event by cloning its PID and adjusting timestamps to preserve causal ordering. The noise pool consists of six neutral patterns such as opening `/dev/null`. By varying the injection ratio λ from 5 to 2000, we expanded the raw trace size by up to 2000×.

As shown in Figure 5, the system remains robust under this extreme stress. Detection accuracy holds at 98% with 1000× noise and remains above 90% even at the 2000× level. Crucially, we observe a cost saturation effect where token consumption decouples from the raw input volume. Despite the linear surge in data, the LLM input size plateaus at approximately 19k tokens for $\lambda \geq 50$. This confirms that our local analyzer imposes a strict computational ceiling and effectively neutralizes flooding attacks.

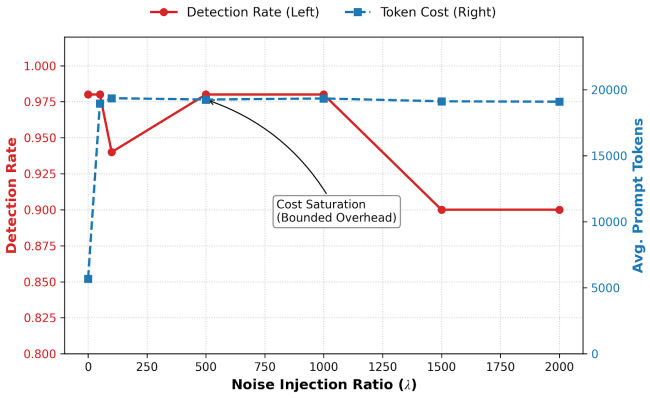

Fig. 5. Robustness under Extreme Noise Injection. Despite noise increasing linearly up to 2000×, Token Cost (blue dashed) plateaus at $\approx 19k$ (bounded overhead), while Detection Rate (red solid) remains robust ($> 90\%$).

4.3 Explanation Quality and Grounding

VirusTotal Consistency Check. We first use a VirusTotal (VT) [44] consistency check as a sanity check against external threat intelligence. Specifically, we extract the predicted family and `threat_category` from each LLM-generated report, and compare them with 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 5). In all malware reports, 790 can be linked to VT metadata. DeepSeek achieves a 90.1%

category match rate, while Qwen achieves 74.7%; for family matching, the rates are 55.6% and 59.0%, respectively. Overall, category-level agreement is higher than fine-grained family token overlap, which is expected given VT aliasing and heterogeneous naming. We therefore treat this check as evidence that our reports capture the right high-level behavior, rather than as a strict ground-truth family labeling.

Table 5. Grounding results (VT consistency and Evidence Match).

Analyzer	Family	Category	Any	Avg. ratio	$\mathbf{SR_{all}}$
DeepSeek V3.2	55.6%	90.1%	93.8%	0.482	80.9%
Qwen Plus	59.0%	74.7%	85.7%	0.334	88.5%

VT rates are computed on 790 reports matched to VT metadata; SR_{all} is computed on 844 reports with traces.

Evidence Match. External labels can be noisy, so we further introduce *Evidence Match* to measure trace-level grounding: whether each `Key Evidence` item in a report can be verified in the corresponding raw JSONL execution trace. For each report, we normalize every evidence bullet into matchable indicators following Table 6, and perform line-level matching against the trace. An evidence item is marked as *supported* if any extracted indicator appears in the trace; otherwise it is *unsupported*. Let E_r be the number of evidence items in report r, and S_r the number of supported items. We compute the overall (micro-averaged) support rate as

$$\mathrm{SR}_{\mathrm{all}} = \frac{\sum_{r \in \mathcal{R}} S_r}{\sum_{r \in \mathcal{R}} E_r}.$$

Across 844 reports, DeepSeek achieves $\mathrm{SR}_{\mathrm{all}} = 80.9\%$, while Qwen reaches 88.5% (Table 5). These results suggest that, although different LLMs may phrase explanations differently, the key evidence in our reports is largely backed by observable runtime artifacts, making the explanations reproducible and easy to audit.

Table 6. 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. The experiments were performed on a host machine deploying our system with a concurrency level of 7 Docker pools. For each container, we enforced strict Linux cgroups limits: a 1 GB memory cap, a 30% CPU quota, a maximum of 1,000 processes (PIDs), a limit of 50 fork operations, and a hard execution timeout of 60 seconds.

We compared MirrorShield against two baselines: (i) a *Native* host, and (ii) the standard `strace` utility (representing traditional `ptrace`-based interception). We deployed a full system emulation using `qemu-system-aarch64`. The virtual machine was configured with a Cortex-A57 vCPU, 512 MB of RAM, and ran Alpine Linux (kernel `vmlinuz-virt`). The emulation was executed in non-graphic mode with standard VirtIO networking.

Lifecycle Agility (Startup & Teardown). We evaluated the system's agility by benchmarking the effective startup and teardown latencies against LiSa [8]. To ensure a fair comparison, we instrumented the orchestration logic of both systems to capture precise timestamps. For MirrorShield, startup latency is measured from the initialization of the Docker provisioning request to the reception of the first system call event by the eBPF probe. 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 completion of the `staprun` instrumentation loader. Teardown latency is consistently defined as the duration from the target program's termination (or execution timeout) to the complete shutdown of the container or VM. Measurements were averaged across distinct trials using diverse samples for each architecture to account for initialization variance. As detailed in Table 7(b), eliminating the overhead of full-system emulation translates into a 4.8× to 14.6× speedup across architectures.

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. [48] Prior sandboxing pipelines often combine `QEMU` emulation with `strace` to obtain cross-architecture executability with syscall-level visibility [49, 30]. Accordingly, we adopt two representative baselines: (i) native `strace` for lightweight syscall tracing, and (ii) `QEMU` user-mode+`strace` as a standard cross-architecture tracing pipeline. We further include LiSa as a research reference, which combines QEMU-based emulation with SystemTap tracing to collect behavioral telemetry. [8] As summarized in Table 7(a), MirrorShield introduces only single-digit syscall overhead across architectures (about 4.4×–9.6×). In contrast, ptrace-based monitoring (`strace` and VM+`strace`) is hundreds to thousands of times more expensive, and the SystemTap-based baseline (LiSa) also incurs substantially higher overhead (tens of times on x86 and much higher under emulation).

End to End Performance and Cost. We evaluated end-to-end throughput on a dataset of $N = 176$ samples. In *Local Mode* (without external API calls),

Table 7. 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	326.7×
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×
QEMU VM (ARM)	1.109	22.63×
VM + `strace`	297.13	6063×

(b) Lifecycle Latency Comparison

Arch	Ours	LiSa [8]	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.

processing completed in 108.82 s. Enabling the DeepSeek API with concurrency = 7 increased the total time to 438.94 s, where the added latency is dominated by remote inference and throttling effects rather than local tracing. During the run, CPU utilization indicates effective parallel execution, with an average of 69.92% and a peak of 308.85% (aggregated across CPU cores). Kernel-mode CPU utilization averaged 24.28%, consistent with overhead from container scheduling and frequent process creation. Memory usage remained stable between 9.41–10.00 GiB, with no sustained upward trend observed throughout the experiment, suggesting no progressive memory growth under our per-container cgroup limits. We also estimate the LLM cost using the provider's token pricing: $0.029 per million input tokens (cache hit), $0.286 per million input tokens (cache miss), and $0.429 per million output tokens. In our experiments, analyzing 100 samples costs approximately $0.29.

5 Case Study

To evaluate the model's robustness, we conducted a detailed case study on sample `bd7a...da81`[2], a Mirai variant that exhibited instability in traditional environments. Table 8 contrasts our LLM-generated analysis with results from industry-standard sandboxes. The sample triggered a runtime panic (`sigaction failed`) in *Joe Sandbox*[3] , resulting in an aborted analysis that relied solely on static YARA matches for a verdict. Similarly, while *Dr. Web vxCube* [50] successfully executed the sample, it provided only generic behavioral tags without specifying the targets or context of these actions [51]

In contrast, our pipeline successfully reconstructed the full attack chain from the available artifacts. Crucially, the LLM surpassed the generic tags of Dr. Web by identifying the specific "Killer" module behavior and accurately listing

[2]Full SHA-256: `bd7a37a86a1fa6831f426d1570f702aaffe454b4609eda5f111b7bd0f4d6da81`
[3]Report available at: sandbox report.

the deleted competitor files. Furthermore, it extracted the precise shell script used for persistence, converting a generic alert into actionable intelligence. This demonstrates the LLM's ability to bridge the gap between low-level execution logs and high-level semantic understanding.

Table 8. Comparative Analysis: Mirai Variant (Sample `bd7a...`)

Commercial (Joe / Dr.Web) *Status: Crash/Success*	**MirrorShield (Ours)** *Status: Success*
Joe Sandbox [21]: CRASHED. Verdict: Generic Malicious. Fallback: Static Yara Detection. **Dr. Web [50]:** 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 enables the lightweight scalability central to MirrorShield, it introduces an inherent trade-off regarding isolation strength. Unlike hardware-level virtualization, containers share the host kernel, creating a larger attack surface for privilege escalation. In practice, container escape vulnerabilities remain a critical concern, ranging from runtime-level flaws like CVE-2019-5736 [52] and CVE-2024-21626 [53], to kernel-specific exploits such as CVE-2022-0492 [54] (cgroups v1 release_agent abuse) and CVE-2022-0847 [55] (Dirty Pipe). To address these risks, MirrorShield actively enforces *User Namespaces (UserNS)*. Empirical studies [56] validate that UserNS effectively neutralizes many critical runtime exploits (e.g., CVE-2019-14271 [57]) by remapping the container's root user to an unprivileged user on the host, serving as a robust containment boundary even when underlying components remain unpatched.

Despite the efficacy of UserNS, strict multi-tenant scenarios may demand stronger isolation. A promising direction for future work is encapsulating containers within *MicroVMs*. This hybrid approach confines the blast radius of kernel escapes to a disposable guest kernel, while still preserving the deployment flexibility that makes containers advantageous over traditional heavyweight VMs.

Limits of Syscall-Centric Analysis. MirrorShield relies on eBPF to capture system calls, which serve as the boundary between user space and kernel space. While highly effective for identifying interactions with the OS (e.g., file I/O, networking), this approach has an inherent *Semantic Gap*. First, purely user-space behaviors such as ransomware encryption loops, string de-obfuscation,

and Domain Generation Algorithms (DGA) execute without invoking system calls. Consequently, these activities remain invisible to our tracer until an I/O operation is triggered. Second, advanced malware may employ *Syscall Evasion* techniques, such as user-mode hooking or utilizing unmonitored legacy system calls to bypass detection. Future iterations of MirrorShield could bridge this gap by integrating *User-Space Probes (uprobes)* to intercept standard library calls, or correlating CPU performance counters to detect computation-heavy malicious logic.

Transparent Artifacts and Evasion Risks. We acknowledge that `qemu-user` introduces detectable artifacts. However, since most IoT malware prioritizes rapid propagation over complex evasion, perfect transparency is rarely required. MirrorShield is therefore optimized for high-throughput triage rather than absolute stealth,as highly evasive samples remain better suited for resource-intensive bare-metal analysis [12]. Future work will focus on scrubbing these emulation fingerprints to further reduce the detection surface.

7 Conclusion

In this work, we presented MirrorShield, a lightweight framework for cross-architecture Linux malware analysis that aims to balance scalability with behavior visibility. MirrorShield executes binaries via containerized QEMU user-mode emulation and collects syscall-level telemetry using an eBPF-based kernel monitor. By combining efficient telemetry collection with local analyzer alerts and evidence-centered trace curation, the framework supports end-to-end analysis and produces readable reports that remain tied to observed runtime artifacts.

We evaluated MirrorShield across 11 CPU architectures and showed strong detection performance (94.7% accuracy, F1: 0.946) with robust generalization, including long-tail architectures observed in the wild. Under extreme trace flooding, detection remained above 90% even with up to 2,000× noise injection. To assess report reliability, we introduced an *Evidence Match* evaluation and found that 80.7% of key report claims can be directly linked to collected trace artifacts. Compared to representative Linux sandbox baselines (QEMU+`strace` and LiSa), MirrorShield reduces analysis turnaround time by over 200× in our evaluation, while maintaining high detection performance.

We acknowledge that container-based execution provides weaker isolation than hardware-level virtualization. In practice, MirrorShield can be deployed on hardened hosts with strict resource limits, or integrated into standard cloud environments where isolation and access control are provided by mature infrastructure. Unlike approaches that require booting a dedicated full-system VM for each target architecture and sample, MirrorShield enables lightweight, multiplexed analysis sessions within a shared runtime, making it suitable for large-scale, cloud-based deployments.

A Appendix A. Full LLM Analysis Report

In this appendix, we present the key part of the analysis report generated by the LLM for sample `bd7a...da81`, as shown in Fig. 6.

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. 6. 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.

B Appendix B.LLM Prompt Templates

The specific prompts designed for the Chain-of-Thought reasoning are detailed below. We utilize a structured prompt containing five sections to guide the LLM's reasoning process.

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 Global Times. *Cyberattacks against DeepSeek escalate with botnets joining, command surging over 100 times: lab*. Accessed: 2025-12-07. Jan. 2025. URL: https://www.globaltimes.cn/page/202501/1327697.shtml.

3 Emanuele Cozzi et al. "Understanding linux malware". In: *2018 IEEE symposium on security and privacy (SP)*. IEEE. 2018, pp. 161–175.

4 Paul Royal et al. "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.

5 Claudio Guarnieri et al. "The Cuckoo Sandbox". In: *Black Hat USA*. Open Source Software. 2012. URL: https://cuckoosandbox.org/.

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

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

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

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

10 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.

11 Xu Chen et al. "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.

12 Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. "BareCloud: Bare-metal Analysis-based Evasive Malware Detection". In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 287–301. ISBN: 978-1-931971-15-7. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/kirat.

13 Tamas K Lengyel et al. "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.

14 J Todd McDonald. *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*. ACM, 2015.

15 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.

16 Fengwei Zhang et al. "Towards Transparent Debugging". In: *IEEE Transactions on Dependable and Secure Computing* 15.2 (2018), pp. 321–335. DOI: 10.1109/TDSC.2016.2545671.

17 Nelson F. Liu et al. "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.

18 Patrick Lewis et al. "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2020.

19 Ilia Shumailov et al. "Sponge Examples: Energy-Latency Attacks on Neural Networks". In: *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2021, pp. 212–231.

20 Fabio Pierazzi et al. "Intriguing Properties of Adversarial ML Attacks in the Problem Space". In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020, pp. 1332–1349.

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

22 Xabier Ugarte-Pedrero et al. "PyREBox: A Python Scriptable Reverse Engineering Sandbox". In: *Black Hat USA*. 2017.

23 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/.

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

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

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

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

28 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.

29 Xin Xiao et al. "Android Malware Detection Based on System Call Sequences and LSTM". In: *Multimedia Tools and Applications* 78 (2019), pp. 3979–3999.

30 Liting Deng et al. "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.

31 Senming Yan et al. "A survey of adversarial attack and defense methods for malware classification in cyber security". In: *IEEE Communications Surveys & Tutorials* 25.1 (2022), pp. 467–496.

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

33 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.

34 Yiling He et al. "On Benchmarking Code LLMs for Android Malware Analysis". In: *Proceedings of the 34th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA Companion '25. Clarion Hotel Trondheim, Trondheim, Norway: Association for Computing Machinery, 2025, pp. 153–160. ISBN: 9798400714740. DOI: 10.1145/3713081.3731745. URL: https://doi.org/10.1145/3713081.3731745.

35 Chongzhou Fang et al. "Large language models for code analysis: Do {LLMs} really do their job?" In: *33rd USENIX Security Symposium (USENIX Security 24)*. 2024, pp. 829–846.

36 Yibiao Wu et al. "A Survey of Machine Learning Approaches for Malware Detection". In: *2025 5th International Conference on Computer Network Security and Software Engineering (CNSSE)*. ACM, 2025. DOI: 10.1145/3732365.3732410.

37 *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.

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

39 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.

40 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.

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

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

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

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

45 DeepSeek. *Models & Pricing (DeepSeek API Docs): DeepSeek-V3.2*. https://api-docs.deepseek.com/quick_start/pricing. Accessed: 2026-02-07. 2025.
46 Alibaba Cloud. *Model list (Model Studio): Qwen-Plus*. https://www.alibabacloud.com/help/en/model-studio/models. Accessed: 2026-02-07. 2025.
47 Hatching International B.V. *Hatching Triage*. Accessed: 2026-01-31. URL: https://hatching.io/triage/.
48 Larry McVoy and Carl Staelin. "lmbench: Portable Tools for Performance Analysis". In: *USENIX Annual Technical Conference*. 1996, pp. 279–294.
49 Hai-Viet Le and Quoc-Dung Ngo. "V-Sandbox for Dynamic Analysis IoT Botnet". In: *IEEE Access* 8 (2020), pp. 145768–145786. DOI: 10.1109/ACCESS.2020.3014891.
50 Doctor Web. *Dr. Web vxCube*. Accessed: 2025-12-14. 2025. URL: https://vxcube.drweb.com/.
51 Dr. Web. *vxCube Analysis: Mirai Variant*. Accessed: 2026-01-27. 2025. URL: https://bazaar.abuse.ch/sample/bd7a37a86a1fa6831f426d1570f702aaffe454b4609eda5f111b7bd0f4d6da81/.
52 NIST National Vulnerability Database. *CVE-2019-5736: runc Container Escape Vulnerability*. https://nvd.nist.gov/vuln/detail/CVE-2019-5736. 2019.
53 NIST National Vulnerability Database. *CVE-2024-21626: runc File Descriptor Leak Vulnerability*. https://nvd.nist.gov/vuln/detail/CVE-2024-21626. 2024.
54 NIST National Vulnerability Database. *CVE-2022-0492: Linux Kernel cgroups release_agent Privilege Escalation*. https://nvd.nist.gov/vuln/detail/CVE-2022-0492. 2022.
55 NIST National Vulnerability Database. *CVE-2022-0847: Linux Kernel Privilege Escalation via Dirty Pipe*. https://nvd.nist.gov/vuln/detail/CVE-2022-0847. 2022.
56 Michael Reeves et al. "Towards Improving Container Security by Preventing Runtime Escapes". In: *2021 IEEE Secure Development Conference (SecDev)*. 2021, pp. 38–46. DOI: 10.1109/SecDev51306.2021.00022.
57 NIST National Vulnerability Database. *CVE-2019-14271: Docker cp Code Injection Vulnerability*. https://nvd.nist.gov/vuln/detail/CVE-2019-14271. 2019.