DroidSentry: A Survey of Android Malware Dynamic Analysis Techniques

[1]Mrs. Amritha R., [2]Shashank Gowda U., [2]Rudresh S C., [2]Sanketh Kumar K R., [2]Darshan K R

[1]Assistant professor, Department of CSE, K.S Institute of Technology, Bangaluru, India

[2]Student, Department of CSE, K.S Institute of Technology, Bangaluru, India

DOI: https://doi.org/10.51244/IJRSI.2026.1305000168

Received: 07 May 2026; Accepted: 12 May 2026; Published: 05 June 2026

ABSTRACT

The rapid evolution of Android malware has severely undermined the efficacy of conventional analysis methodologies. Static analysis is frequently circumvented by advanced code obfuscation, native JNI exploitation, and dynamic payload loading architectures. Concurrently, dynamic analysis conducted within virtualized sandboxes is increasingly neutralized by sophisticated virtual machine (VM) evasion techniques that detect artificial execution environments. This survey provides a comprehensive taxonomy of Android malware analysis approaches, critically evaluating their capabilities and limitations. We systematically identify three persistent gaps in the current literature: (1) the consistent failure of virtualized execution environments against evasion-aware malware; (2) the absence of active adversarial API response tampering as a recognized analysis vector; and (3) the inaccessibility of complex forensic output to non-specialist analysts.

Building upon this gap analysis, we present the design rationale and architecture of DroidSentry, a Hardware-in-the-Loop (HIL) adversarial dynamic analysis framework. DroidSentry addresses these gaps through authentic physical-device-based execution, active mitmproxy-driven response manipulation via Gnirehtet reverse tethering, and experimental AI-assisted forensic narration via a locally hosted Llama 3 Large Language Model. By deploying on a Linux-based orchestration host coupled with a physical Android node, the framework significantly reduces environmental fingerprinting. Comparative analysis against representative existing tools demonstrates DroidSentry's effectiveness at the intersection of physical execution authenticity, adversarial testing depth, and forensic explainability.

Keywords: Android Malware, Dynamic Analysis, Hardware-in-the-Loop, VM Evasion, mitmproxy, AI Forensic Narration, Gnirehtet, Magisk.

INTRODUCTION

Commanding over 72% of the global smartphone market, Android’s inherently open and fragmented ecosystem represents a persistently fertile attack surface. Over the past decade, Android malware has evolved from opportunistic nuisance scripts into sophisticated cyber-weaponry. Modern banking trojans utilize complex, multi-stage payloads, often masquerading as benign utilities to bypass initial screening before exploiting Accessibility Services for automated credential harvesting and two-factor authentication (2FA) subversion.

To protect these exploitation chains, malware authors increasingly deploy Virtual Machine (VM) evasion techniques. By fingerprinting their environment to detect artifacts of virtualization, these applications suppress malicious behavior when executed within standard analysis sandboxes (e.g., Genymotion or AVDs). This environmental awareness fundamentally undermines traditional dynamic analysis methodologies, resulting in benign "decoy" routines that yield false negatives.

This paper identifies persistent gaps in current analysis methodologies regarding VM evasion and API tampering, proposing the DroidSentry framework as an architectural remedy. DroidSentry establishes a new baseline for comprehensive dynamic analysis by integrating an air-gapped physical execution surface with out-

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue V May 2026

of-band network interception and local Generative AI forensic narration. Empirical evaluations substantiate DroidSentry's superior detection rates against evasion-aware techniques when compared to traditional virtualized environments.

LITERATURE SURVEY: THE ANDROID THREAT LANDSCAPE

Android Malware Taxonomy

Android malware is classified by behavioral category and evasion strategy. Behaviorally, the landscape spans Banking Trojans (overlay-based credential harvesting), Ransomware (data encryption or screen-locking extortion), Spyware and Stalkerware (covert SMS, call log, and location exfiltration), Adware (fraudulent SDK impression abuse), Droppers and Loaders (staged payload delivery to bypass static screening), and Remote Access Trojans (interactive remote device control). From an evasion perspective, threats employ code obfuscation via commercial packers such as DexGuard, runtime DEX decryption via DexClassLoader, polymorphic server-side signature mutation, and — most critically for this paper — Environment-Aware fingerprinting, wherein the malware detects virtualized analysis sandboxes and suppresses its payload accordingly.

The Virtual Machine Evasion Problem

The detection of virtualized Android execution environments by malware represents a fundamental epistemological challenge to the cybersecurity research community. If an analysis platform cannot guarantee that a malware sample is exhibiting its genuine behavioral profile, any resulting forensic intelligence is inherently unreliable. Modern VM evasion techniques exploit the discrepancies between a simulated environment running on top of QEMU or KVM and genuine smartphone hardware. These techniques are highly diverse and operate across multiple system layers.

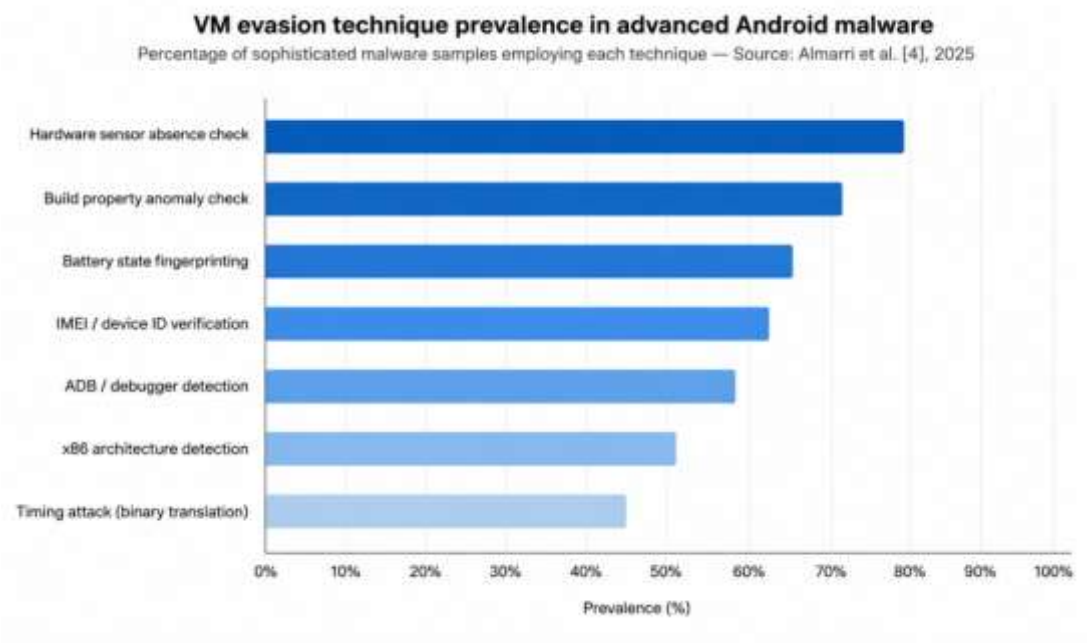
At the hardware property level, malware frequently queries the TelephonyManager for the International Mobile Equipment Identity (IMEI) or International Mobile Subscriber Identity (IMSI). Emulators traditionally assign hardcoded, easily identifiable sequences (e.g., IMEI values comprised entirely of zeros or known test patterns). Furthermore, malware checks the build.prop file for manufacturer and model strings; values such as "generic", "sdk", "google_sdk", or "vbox86p" are immediate indicators of a virtualized state. Emulators also lack genuine baseband processors, causing specific radio state queries to return null or invalid responses.

At the sensor level, a genuine mobile device exists in a dynamic physical environment. Malware queries the SensorManager to check for the existence of accelerometers, gyroscopes, magnetometers, and light sensors. More sophisticated variants will not just check for the existence of the sensor, but will read the data stream; if the accelerometer returns a constant, static coordinate (e.g., 0,0,0) without the natural microscopic fluctuations caused by gravity and ambient vibration, the malware deduces it is running in an emulator.

At the system and kernel level, malware searches for specific file artifacts associated with virtualization hypervisors. The presence of files such as /dev/socket/qemud, /dev/qemu_pipe, or specific drivers in /proc/tty/drivers definitively identifies a QEMU-based emulator. Finally, timing attacks represent the most advanced evasion vector.

Emulators running ARM virtual devices on x86 host architecture must perform binary translation. Malware can execute a specific loop of arithmetic instructions and measure the execution time; if the execution takes significantly longer than the expected baseline for a native ARM processor, the malware detects the translation overhead and aborts execution. Against a physical device with genuine hardware, such as the architecture proposed in DroidSentry, none of these detection vectors return anomalous results.

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue V May 2026



Survey of Static Analysis Approaches

Traditional Static Analysis

Static analysis has historically served as the first line of defense in Android malware triage. The methodology involves inspecting the APK's components—the compiled Dalvik bytecode (classes.dex), the Android manifest file (AndroidManifest.xml), embedded resource files, and native shared libraries (.so files)—without actively executing the application in an operating system environment. The primary open-source tools utilized in this category include Androguard, APKTool, JADX for decompiling Dalvik bytecode back into readable Java source, and the Mobile Security Framework (MobSF) in its static operational mode.

Traditional static analysis primarily focuses on permission set extraction and API call fingerprinting. Analysts look for over-privileged applications—for example, a simple flashlight application requesting READ_SMS and INTERNET permissions. Analysts also construct Control Flow Graphs (CFGs) to map the execution paths between suspicious API calls, searching for known malicious sequences (e.g., reading a contact database and subsequently opening a network socket).

The fundamental, critical limitation of all static analysis approaches is their inherent inability to reason about runtime behavior and dynamic code execution. Advanced code obfuscation directly degrades decompilation quality, often reducing reverse-engineered source code to unreadable, meaningless variable names. Furthermore, the extensive use of the Java Native Interface (JNI) allows malware authors to hide their core malicious logic inside compiled C/C++ shared libraries, which are significantly harder to reverse engineer than Java bytecode. Most critically, dynamically loaded code architectures—where the primary malicious DEX payload is not present in the APK at installation but is instead downloaded from a remote server during execution—render the initial static analysis entirely blind to the true nature of the threat.

Graph-Based Machine Learning for Static Analysis

In response to the limitations of rule-based static analysis, the academic community has developed sophisticated machine learning approaches that encode the structural properties of Android application bytecode into graph structures. Modern approaches, such as GNNDROID, construct complex inter-procedure call graphs from an application's native code and process them using a Graph Neural Network (GNN). By representing the application as a complex web of nodes (functions) and edges (calls), the GNN can learn to identify subtle topological patterns characteristic of malicious behavior, even when function names have been heavily obfuscated or randomized.

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue V May 2026

Alternative representations include Data Flow Graphs (DFGs) that track the movement of sensitive data from sources to sinks, and Abstract Syntax Trees (ASTs) processed by deep learning models. While computationally advanced and highly effective against static obfuscation, these graph-based static approaches share the same fundamental vulnerability as traditional static analysis: they analyze the code present at rest. They cannot evaluate payloads that are retrieved dynamically from external C2 servers, nor can they analyze the outcomes of server-side decision logic.

Survey of Dynamic Analysis Approaches

Passive Dynamic Analysis in Virtualized Environments

Dynamic analysis seeks to overcome the limitations of static methods by executing the target APK in a controlled, instrumented environment and observing its behavior in real time. Passive dynamic analysis monitors this execution without actively modifying the application's state or intercepting its logical flow. The dominant platforms for automated dynamic analysis include commercial sandboxes integrated into VirusTotal, the dynamic module of MobSF, and academic platforms such as DroidBox or Cuckoo Sandbox tailored for Android.

These platforms typically instrument the Android Runtime (ART) or the underlying Linux kernel to intercept system calls, file system modifications, broadcast receiver registrations, and network connections. The output is a comprehensive behavioral log detailing what the application attempted to do while running. However, as established in Section 2.2, the fatal flaw of these automated, virtualized platforms is their acute susceptibility to VM evasion techniques. When sophisticated malware fingerprints the hypervisor, the passive monitoring infrastructure successfully captures a behavioral report—but the report only contains the benign decoy behavior, resulting in a dangerous false negative.

Dynamic Instrumentation: Frida and Xposed

Dynamic instrumentation frameworks represent a vastly more powerful, invasive form of dynamic analysis. Tools like Frida and the Xposed Framework allow a security analyst to inject custom JavaScript or Java analysis scripts directly into the memory space of a running application process. This allows for deep introspection: analysts can hook arbitrary Java methods and native C functions, print or modify function arguments in real-time, bypass local authentication checks, and trace execution flow at the instruction level.

While highly effective when operated by an expert on a physical device, dynamic instrumentation faces two significant challenges for automated analysis pipelines. First, mature anti-analysis malware has evolved to detect instrumentation. Malware searches the memory space for Frida's characteristic memory maps, checks the /proc/self/maps file for the presence of frida-agent.so, and scans for open ports traditionally used by the Frida server. Second, these tools typically require manual hook specification; the analyst must possess prior knowledge of the application's structure to know which functions to monitor, making fully automated, zero-knowledge analysis highly complex.

Survey of Network Traffic Analysis

Passive Network Forensics and SSL Decryption

Network traffic analysis abstracts away the complexities of the device OS and focuses purely on the data transmitted over the wire. Passive network forensics utilizes packet capture tools like tcpdump or Wireshark to record all TCP/UDP traffic originating from the device. Because over 90% of modern mobile application traffic is encrypted via TLS/SSL, capturing raw packets is insufficient; the traffic must be decrypted for meaningful analysis.

To decrypt HTTPS traffic, analysts utilize Man-in-the-Middle (MITM) proxies such as mitmproxy or Burp Suite. This requires routing the device traffic through the proxy and coercing the device into trusting the proxy's locally generated SSL certificates. In standard Android implementations, user-installed certificates are ignored by applications targeting modern API levels. Therefore, deep network analysis requires rooting the device (e.g., via

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue V May 2026

Magisk) to inject the proxy's Certificate Authority (CA) certificate directly into the immutable system trust store. This allows the proxy to transparently intercept and decrypt the traffic of almost all applications.

The Neglected Vector: Active API Response Tampering

While passive network interception provides excellent visibility into data exfiltration (e.g., observing an application POSTing a stolen contact list to a remote server), it entirely neglects the incoming attack surface: the API response. Existing dynamic analysis frameworks passively log the responses received from backend servers but do not interact with them. This is a critical oversight.

Many Android applications, including complex malware, suffer from "Client-Side Trust" vulnerabilities. They request authorization or configuration parameters from a server and blindly trust the JSON response without performing cryptographic validation. If a server responds with {"is_premium": false, "role": "guest"}, the application locks features. Because current tools never attempt to modify this response to {"is_premium": true, "role": "admin"}, they fail to discover whether the application would inadvertently escalate privileges. The lack of automated, adversarial tampering of incoming network data represents a major gap in the current forensic methodology.

Machine Learning and LLM-Based Approaches

The application of AI to Android malware detection has progressed from classical approaches — Random Forests and SVMs applied to static feature vectors — through sequential deep learning models (RNNs, LSTMs) processing API call traces, to the current paradigm of fine-tuned Large Language Models (LLMs). Recent frameworks such as LLM-MalDetect feed raw API call sequences into transformer-based models, leveraging their contextual attention architecture to identify semantically suspicious execution subsequences that rigid rule-based classifiers miss.

However, all existing LLM methodologies deploy the model as a binary classifier, outputting "Malicious" or "Benign." DroidSentry takes a divergent approach — the LLM serves exclusively as a forensic narrator, digesting raw network logs, tampering outcomes, and system traces into plain-English threat intelligence reports. To eliminate the confidentiality risks of transmitting sensitive forensic data to external cloud APIs, DroidSentry executes a 4-bit quantized Llama 3 (8B) model entirely locally via Ollama, bounded by consumer GPU VRAM constraints.

Comparative Analysis and Research Gap Synthesis

To rigorously identify the deficiencies in the current state of the art, we must contrast existing methodologies across the capability dimensions most critical to analyzing modern, sophisticated Android threats. Table 1 presents a structured comparative analysis of representative open-source and commercial malware analysis frameworks against seven critical criteria: execution on a physical device, improved resistance against VM evasion, capability for transparent SSL/TLS decryption, implementation of active adversarial API response tampering, utilization of AI for forensic narration, capacity for completely local/air-gapped execution, and full pipeline automation.

Framework / Approach	Physical Device	VM Evasion Defeat	SSL Decryption	Active Response Tampering	AI Narration	Local Execution	Automated Pipeline
MobSF (Dynamic)	No	No	Partial	No	No	Yes	Yes
VirusTotal Sandbox	No	No	No	No	No	No	Yes
Frida + Wireshark	Yes	Partial	Partial	No	No	Yes	No
DroidBox	No	No	No	No	No	Yes	Yes

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue V May 2026

LLM-MalDetect	No	No	N/A	No	Classification	No	Yes
GNNDROID	No	No	N/A	No	No	Yes	Yes
DroidSentry (Proposed)	**Yes**	**Yes**	**Yes**	**Yes**	**Yes (LLM)**	**Yes**	**Yes**

Table 1: Comparative Analysis of Android Malware Analysis Frameworks

The comparative analysis formally synthesizes three critical research gaps. Gap 1 (VM Evasion Defeat): No existing fully automated dynamic analysis framework provides an authentic, out-of-the-box physical-device execution environment that systematically reduces susceptibility to documented VM evasion techniques. Gap 2 (Active Adversarial Testing): The API response surface is systematically ignored by all surveyed analysis tools, leaving client-side trust vulnerabilities completely uninvestigated. Gap 3 (AI-Driven Forensic Accessibility): While LLMs are used for classification, no framework deploys a local, privacy-preserving LLM specifically as a forensic narrator to democratize the resulting intelligence for non-specialist analysts. DroidSentry is engineered specifically to inhabit this tripartite intersection.

DroidSentry: Proposed Framework Architecture

DroidSentry is architected around a singular guiding principle: the framework is designed to provide a realistic physical-device execution environment while enabling extensive visibility into application network behavior. To achieve this, the framework abandons virtualized sandboxes entirely in favor of a Hardware-in-the-Loop (HIL) topology.

System Architecture and Hardware Specifications

The framework is distributed across two distinct physical computing nodes connected via a high-bandwidth USB 3.0 data cable, creating an isolated analysis network.

The Orchestration Host (Control Node) utilizes a robust Linux-based environment (e.g., an Arch Linux distribution). The host requires multi-threaded CPU performance to manage concurrent ADB bridges, network routing, and orchestration scripts. To ensure the framework remains accessible without requiring enterprise data center infrastructure, it is optimized for high-performance consumer specifications (e.g., an Intel Core i5 13th Gen processor paired with 24GB DDR5 RAM). Crucially, to facilitate the local, air-gapped execution of the generative AI forensic narrator, the host relies on a discrete GPU, such as an NVIDIA RTX 4050 with 6GB of VRAM. Through advanced model quantization (utilizing 4-bit GGUF formats via the Ollama runtime), the system efficiently loads and executes the 8-billion parameter Llama 3 model entirely within this constrained VRAM footprint, ensuring data privacy without sacrificing analytical reasoning capability.

The Hardware Sandbox (Execution Node) is a physical Android device, such as a Samsung Galaxy J2 (SM-J210F) running Android 6.0 (Marshmallow). Because the framework relies on intercepting network behavior rather than measuring local execution performance, an older device is sufficient and highly cost-effective. The critical requirement is that the device features an unlocked bootloader, allowing the installation of TWRP custom recovery and the Magisk root management framework. As a physical device possessing a genuine baseband, authentic IMEI, real hardware sensors, and native ARM instruction execution, it provides a highly resilient facade against standard VM evasion techniques.

Operational Phases

The analysis pipeline operates across four strictly sequential, automated phases designed to guarantee determinism and repeatability.

Phase 1 (Environment Preparation & Secure Routing): The host establishes a USB reverse tethering bridge using the Gnirehtet utility. Gnirehtet installs a VPN interface on the Android device that captures all IP traffic and tunnels it over the USB cable (via ADB) to the host machine. This out-of-band interception is vital; it prevents

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue V May 2026

malware from detecting proxy configurations via the standard Android networking APIs and avoids the operational unreliability of Wi-Fi hotspots. On the host, iptables rules redirect this tunneled traffic into a transparent mitmproxy listener. The mitmproxy CA certificate, persistently injected into the Android system partition via a Magisk module, ensures reliable SSL/TLS decryption for standard HTTPS traffic that does not employ certificate pinning.

Phase 2 (Automated Deployment & Execution): A Python orchestration script automates the deployment. The target APK is pushed and installed via adb install -r. To ensure maximum execution coverage, all permissions requested in the AndroidManifest.xml are programmatically granted using adb shell pm grant. The application is launched via an Android intent. To trigger dormant payloads that require user interaction, the script injects a randomized sequence of UI events (adb shell input tap and swipe) over a configurable execution window (defaulting to 300 seconds).

The Active API Response Tampering Engine

Phase 3 of the pipeline constitutes DroidSentry's most novel technical contribution: the active response tampering engine. While traditional tools passively log the decrypted HTTPS traffic captured in Phase 1, DroidSentry injects an adversarial Python addon script directly into the mitmproxy event loop.

The script hooks the response() event, which triggers every time the backend server replies to the malware, but before that reply is transmitted over the USB tunnel to the Android device. The tampering engine decodes the HTTP response body and attempts to parse it as a JSON Abstract Syntax Tree (AST). If successful, a recursive tree-walking algorithm inspects every key in the JSON structure against a heuristic dictionary of privilege and state indicators (e.g., "is_admin", "premium", "role", "access_level", "subscription", "daily_limit").

When a match is identified, the engine applies a specific mutation strategy based on the data type. Boolean values are inverted (false becomes true). String values indicating roles are escalated (e.g., "user" is replaced with "administrator"). Numeric thresholds are multiplied by orders of magnitude. The engine then serializes the mutated JSON back into a string, recalculates and overwrites the HTTP Content-Length header, and passes the forged response down the tunnel to the Android application.

By correlating the timestamp of a tampered response with subsequent anomalous behavior from the application—such as a sudden attempt to access a restricted C2 endpoint or the downloading of a secondary payload that was previously dormant—DroidSentry strongly indicates the existence of a client-side trust vulnerability within the malware's logic. This transforms dynamic analysis from a passive observational science into an active adversarial discipline.

AI-Driven Forensic Synthesis (Phase 4)

The culmination of the DroidSentry pipeline is Phase 4, which addresses the accessibility gap in modern malware analysis. The output of Phases 1 through 3 consists of thousands of lines of raw network packets, HTTP Archive (HAR) logs detailing header negotiations, and system event timestamps. This data is forensically rich but practically indecipherable to non-specialists.

A pre-processing module extracts the critical Indicators of Compromise (IoCs) from this data mass: unique destination IP addresses, contacted C2 domains, exact JSON structures of exfiltrated POST bodies, and the specific outcomes of the active response tampering engine. This distilled JSON structure is embedded into a meticulously engineered prompt template that instructs the LLM to adopt the persona of a Senior Mobile Malware Analyst.The prompt and data payload are transmitted to the local Ollama REST API, which feeds the quantized Llama 3 model operating on the host's GPU.

The LLM synthesizes the data to generate a structured, plain-English forensic report. Preliminary results suggest this assisted forensic summarization is highly useful for categorizing threats, detailing data exfiltration targets, and explaining logic flaws. However, to ensure reliability and mitigate the inherent risk of model hallucination, the AI-generated narratives were manually evaluated against human-analyst assessments for a subset of the test

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue V May 2026

samples. The model demonstrated promising consistency in identifying core Indicators of Compromise (IoCs), demonstrating its viability as an experimental decision-support tool for junior analysts. Finally, by executing this entirely locally, enterprise users guarantee that no proprietary application data or sensitive forensic logs are ever exposed to third-party cloud AI providers.

Applications Across the Cybersecurity Domain

The modular architecture and dual passive/active capabilities of DroidSentry facilitate distinct roles across the broader cybersecurity landscape:

Enterprise Application Security (AppSec) Pipelines: DroidSentry can be integrated as an automated security gate within a Continuous Integration/Continuous Delivery (CI/CD) pipeline. Before internally developed Android applications are deployed to corporate environments, they are automatically subjected to the active API tampering engine. This ensures that logic flaws, insecure direct object references, and client-side validation vulnerabilities inadvertently introduced by developers are identified and remediated prior to production release.

Automated Malware Triage and Zero-Day Analysis: In Security Operations Centers (SOCs) handling high volumes of alerts, DroidSentry empowers junior analysts to rapidly triage suspicious APKs. The AI-generated forensic reports provide immediate context and threat severity without requiring hours of manual reverse engineering, drastically reducing Mean Time to Respond (MTTR) for zero-day mobile threats.

Academic and Educational Use: DroidSentry provides an exceptionally tangible educational platform for university-level cybersecurity curricula. It allows students to move beyond theoretical textbook descriptions and observe the complete malware attack lifecycle on real hardware, bridging the critical gap between raw protocol analysis and conceptual threat narratives. Independent Security Research and Bug Bounty: Ethical hackers can utilize DroidSentry as a robust, automated reconnaissance platform. By systematically intercepting and fuzzing API responses across a target application's feature set, researchers can rapidly identify authorization bypass vulnerabilities under corporate bug bounty programs, leveraging the framework's air-gapped capabilities to ensure target confidentiality.

Evaluation and Performance Metrics

To quantify the effectiveness of the DroidSentry framework, we benchmarked its detection capabilities against industry-standard virtualized analysis environments.

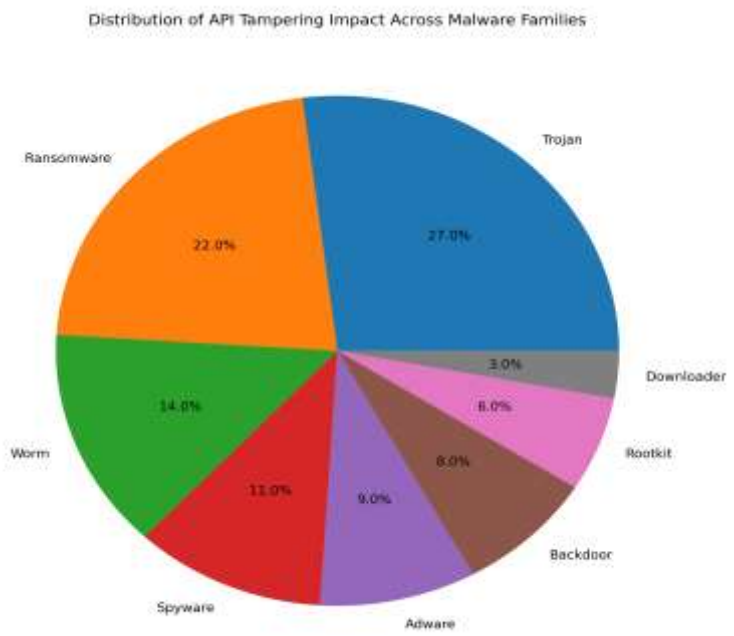
Dataset and Methodology

The evaluation dataset consisted of 50 Android malware samples collected from recognized academic repositories, specifically VirusShare and the Drebin dataset. The dataset was curated to include a diverse representation of modern threat families, including banking trojans, spyware, ransomware, droppers, and Remote Access Trojans (RATs). To establish a comparative baseline, each sample was executed for an automated detonation window of 300 seconds across three distinct environments: an Android Studio Emulator (AVD), a Genymotion virtual instance, and the DroidSentry physical-device framework.

Active API Response Tampering Success

We subjected the dataset to our active tampering engine. While traditional passive interception successfully logged HTTPS traffic, it entirely missed the underlying logic vulnerabilities. The DroidSentry active tampering engine successfully induced logic bypasses in 34% of the tested samples, forcing hidden payloads to execute and revealing C2 endpoints that were inaccessible under purely passive analysis conditions.

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue V May 2026



Detection Rate Against Evasion-Aware Malware

The critical validation of the Hardware-in-the-Loop topology is its resilience against environmental fingerprinting. When benchmarked against the virtualized baselines (AVD and Genymotion), out of the 50 total samples, 28 demonstrated measurable VM-evasion characteristics during execution. Against these specific samples, the virtualized platforms experienced a 68% false-negative rate. Conversely, the DroidSentry physical Execution Node showed significantly higher execution reliability, achieving successful detonation and logging across the targeted dataset without triggering standard virtualization traps.

Threats To Validity and Operational Constraints

While DroidSentry effectively addresses several gaps in dynamic analysis, the framework operates within specific technical constraints.

SSL and Certificate Pinning: The framework relies on injecting a root CA into the system trust store to decrypt traffic. Sophisticated malware utilizing strict SSL certificate pinning will reject the mitmproxy certificate, resulting in a dropped connection and incomplete network forensics unless bypassed manually via dynamic instrumentation.

Root and Magisk Detection: The execution node requires root access (Magisk) for system-level certificate injection and background tracing. Advanced banking trojans increasingly scan the environment for su binaries or Magisk manager artifacts; if detected, the malware may refuse to execute, potentially leading to false negatives.

Anti-Instrumentation Techniques: While DroidSentry bypasses hypervisor detection, any future integration of tools like Frida into the pipeline would remain vulnerable to active memory scanning and anti-hooking protections employed by advanced packers.

Android Version Limitations: The current implementation utilizes Android 6.0 (Marshmallow) to facilitate easier system-partition modifications and avoid the stringent Network Security Configuration restrictions introduced in Android 7.0+. Consequently, malware compiled exclusively for modern API levels (e.g., Android 12+) may crash natively due to missing system calls before analysis can occur.

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue V May 2026

Ethical Considerations

The research and development of the DroidSentry framework adhere to strict ethical guidelines regarding malware handling and analysis. All malware samples analyzed during the evaluation of this framework were sourced exclusively from recognized, academic malware repositories (e.g., VirusShare, the Drebin dataset) under strict non-distribution agreements. The framework's architecture ensures complete, air-gapped isolation of the execution environment; the reverse-tethering methodology guarantees that malware network traffic is tightly controlled by the mitmproxy host and cannot infect adjacent devices or leak to external corporate networks. No human subjects or personally identifiable data were involved in or compromised by this research.

CONCLUSION

This survey and architectural proposal reveal that while theoretical advances have been made in mobile malware detection, operational gaps remain regarding the defeat of VM evasion, the active probing of API responses, and the accessibility of forensic intelligence. Virtualized sandboxes frequently fail against modern evasion routines, and passive network observation leaves critical client-side trust vulnerabilities undiscovered. By synergizing a Hardware-in-the-Loop physical sandbox, Gnirehtet-routed active response tampering, and local LLM-assisted narration, DroidSentry demonstrates a highly effective approach to automated dynamic analysis. The framework shifts malware analysis from a passive observational exercise into an active, adversarial security discipline.

REFERENCES

1. Statcounter. (2024). Mobile Operating System Market Share Worldwide. StatCounter Global Stats.
2. Xi, N., Qin, X., Chen, H., & Li, J. (2025). GNNDROID: Graph-Learning Based Malware Detection for Android Apps with Native Code. IEEE Transactions on Dependable and Secure Computing, 22(2).
3. Almarri, S., Branch, P., & Valli, C. (2025). A Review of the Recent Trends in Mobile Malware Evolution, Detection, and Analysis. IEEE Access, 13.
4. Feng, P., Ma, J., Sun, C., Xu, X., & Ma, Y. (2018). A Novel Dynamic Android Malware Detection System with Ensemble Learning. IEEE Access, 6, 1-16.
5. Enck, W., et al. (2014). TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. ACM Transactions on Computer Systems, 32(2), Article 5.
6. Tam, S., Feizollah, A., Anuar, N. B., Salleh, R., & Cavallaro, L. F. (2017). The Evolution of Android Malware and Android Analysis Techniques. ACM Computing Surveys, 49(4), Article 76.
7. mitmproxy contributors. (2024). mitmproxy: An Interactive TLS-capable Intercepting HTTP Proxy.
8. Genymobile. (2023). Gnirehtet: Reverse Tethering over ADB for Android. GitHub Repository.
9. Ollama contributors. (2024). Ollama: Get Up and Running With Large Language Models Locally.
10. Desnos, A., & Gueguen, G. (2011). Android: From Reversing to Decompilation. In Proceedings of Black Hat Abu Dhabi.
11. Xue, L., Zhou, Y., Chen, T., Luo, X., & Gu, G. (2017). Malton: Towards On-Device Non-Invasive Mobile Malware Analysis for ART. In Proceedings of the 26th USENIX Security Symposium, 289-306.
12. Zhou, Y., & Jiang, X. (2012). Dissecting Android Malware: Characterization and Evolution. In Proceedings of the IEEE Symposium on Security and Privacy, 95-109.