Poster: Reconstructing the Provenance of Android

Mingxiang Shi†, Xiangmin Shen*, Yuqiao Gu†, Zhipeng Chen†, Lingzhi Wang‡, Yi Jiang†, Zhenyuan Li†✉
†Zhejiang University, *Hofstra University ‡Northwestern University

***Abstract*—Modern Android attacks span application code, native libraries, Binder IPC, and kernel-level activities, making it difficult to reconstruct end-to-end causality from a single telemetry source. In this paper, we introduce ARGUS, a runtime provenance system that captures cross-layer causal dependencies by combining kernel-level eBPF monitoring with targeted user-space profiling of selected Java APIs and Android native functions. ARGUS preserves Android-specific semantics by extracting Binder service contexts and correlating caller and callee processes, subsequently merging these insights with low-level file and network events to generate a unified provenance graph. We provide preliminary measurements on representative workloads and a case study demonstrating that ARGUS can link sensitive actions executed via Android system services to downstream effects, relationships that are often fragmented when relying solely on Java-level monitoring.**

I. Introduction and Background

Android devices host a vast ecosystem of third-party applications while simultaneously storing sensitive personal and financial data; consequently, failures in security monitoring can have severe real-world consequences. Many real-world attack workflows span multiple execution layers, rendering incident investigation difficult without cross-layer telemetry. In a typical attack chain, an application may trigger actions via Java APIs, execute critical steps within native libraries, invoke Android system services through Binder IPC, and finally generate kernel-level events such as file I/O or network connections [2]. When telemetry is limited to a single layer, investigators frequently observe low-level events but cannot reliably attribute them to the high-level Android actions or specific system services that initiated them.

This challenge stems from Android's architectural design. Applications operate within isolated sandboxes and access privileged functionality through the Android framework and system services. Many sensitive operations are executed by system services on behalf of applications, with cross-process requests transmitted via Binder IPC. Consequently, an identical kernel-level event can carry vastly different security implications depending on which service interface was invoked and which calling application initiated the request.

Existing telemetry approaches provide only fragmented visibility. While kernel-level tracing offers broad coverage across all processes, it often lacks Android-specific context, particularly regarding Binder transactions and high-level API intent. Conversely, user-space instrumentation captures rich Java-level semantics but may fail to monitor privileged or system processes; it is also susceptible to evasion when attackers shift behavior to native code or bypass hooked paths. Furthermore, strict resource constraints on mobile devices limit the extent of instrumentation that can be sustained continuously.

ARGUS bridges this gap by enhancing provenance visibility on Android through a hybrid approach that combines eBPF-based acquisition with selective user-space semantics. eBPF enables safe, programmable kernel-level monitoring with extensive process coverage and minimal deployment overhead, rendering it suitable for always-on collection on production devices. ARGUS integrates this kernel-level evidence with targeted profiling of specific Java APIs and native functions to construct a unified provenance graph. This holistic view facilitates a wide range of security and forensics tasks, including incident investigation, malware analysis, and the support of downstream detection and response mechanisms [3].

II. System Design

As illustrated in Fig. 1, ARGUS comprises three core components: cross-layer event collection, provenance graph construction, and rule-based threat detection.

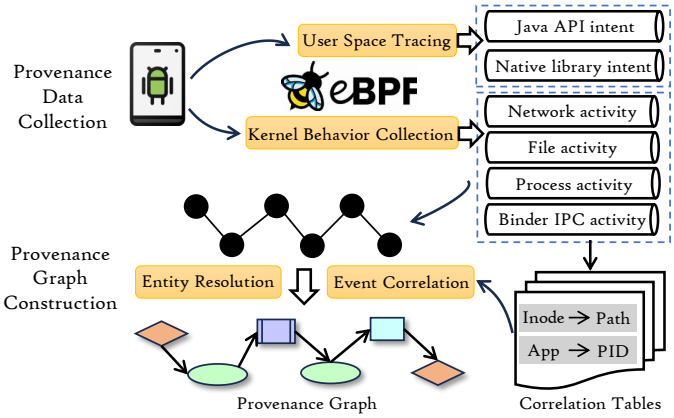


Fig. 1: System Design

Cross-layer provenance data collection. ARGUS aggregates data from both kernel and user spaces. Kernel-level eBPF monitoring ensures broad coverage of system-wide activities, including process execution, file operations, network traffic, and Binder IPC. Complementing this, user-space profiling captures specific Java API and native function calls to recover high-level intent that is often obfuscated in raw kernel events. This hybrid approach is essential, as security-critical actions on Android frequently involve cross-process service invocations via Binder, the semantics of which are not fully recoverable from system calls alone.

Provenance graph construction. ARGUS synthesizes collected events into a unified provenance graph by establishing explicit causal links often missing in raw logs. For file artifacts, operations are correlated using unique inode and device identifier pairs, preserving causal continuity even when file paths

are absent or modified. For Binder transactions, the system links cross-process requests by resolving the target service process and extracting the interface descriptor; this explicitly maps system service usage within the graph. The resulting structure interconnects processes, files, network sockets, and system services, enabling analysts to trace causal paths across execution layers and process boundaries effectively.

III. Preliminary Evaluation

A. Performance analysis

We assess the resource consumption of ARGUS under varying workloads on a Google Pixel 8 smartphone running Android 16. Specifically, we evaluate the system overhead introduced by ARGUS in two representative scenarios: (i) a low-load, interactive usage scenario involving continuous engagement with the TikTok application, and (ii) a high-load stress scenario utilizing the AnTuTu benchmark, which generates intensive file I/O and network connection operations.

As illustrated in Fig. 3, our results indicate that ARGUS incurs an average additional CPU overhead of 15.9% in the low-load scenario and 12.5% in the high-load scenario, suggesting that the relative CPU cost of ARGUS remains stable even under heavy system load. Regarding memory usage, ARGUS introduces an additional footprint of 297.83 MB in the low-load scenario and 766.7 MB in the high-load scenario. Given that high-stress workloads naturally trigger a surge in log generation and requisite buffering, this increase is proportional to the event volume and remains within acceptable limits for modern devices. Overall, these results demonstrate that ARGUS facilitates efficient, low-intrusion acquisition of Android provenance data.

B. Case Study

We present a case study to demonstrate how ARGUS enhances visibility for multi-stage Android attacks. In this scenario, an adversary maintains a Command and Control (C2) channel via a malicious process, `Spynote`. Upon receiving remote commands, `Spynote` conducts reconnaissance by reading `/system/build.prop` before downloading an additional payload, `malicious.dex`, for subsequent execution. The adversary then exploits a privilege escalation vulnerability to compromise `system_server`, thereby gaining access to protected system services and sensitive application data. Once executing within the context of `system_server`, the attacker establishes a local database, `steal.db`, to aggregate stolen information. They subsequently extract account records from the Weibo application, coerce `camera_server` to capture unauthorized photos, store these artifacts in `steal.db`, and finally exfiltrate the database content to remote infrastructure. ARGUS effectively constructs a provenance graph that interconnects these disparate stages through explicit causal edges spanning processes and execution layers.

We compare ARGUS with Frida [1] to delineate the visibility scope of each approach. While Frida effectively captures Java-level behaviors within instrumented application processes, it fails to provide the necessary kernel-level evidence and Binder context to correlate activities across `Spynote`, `system_server`, and `camera_server`. Consequently, Frida yields only fragmented insights into the attack workflow, lacking the cross-process causal linkages required to elucidate how privileged system services mediate access to sensitive data and device resources.

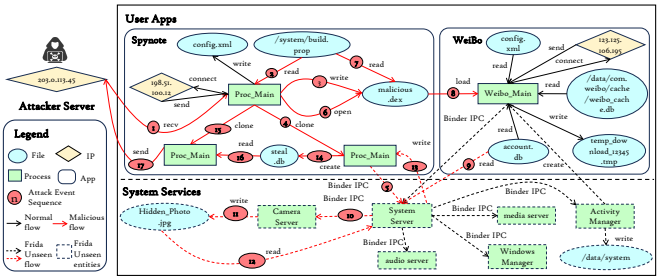
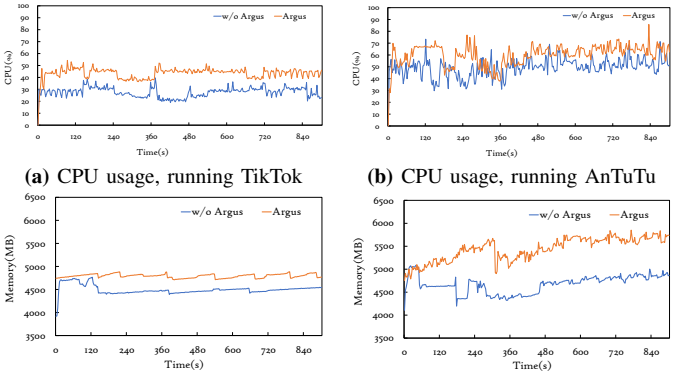


Fig. 2: Provenance graph generated by ARGUS



(a) CPU usage, running TikTok **(b)** CPU usage, running AnTuTu

(c) Memory usage, running TikTok **(d)** Memory usage, running AnTuTu

Fig. 3: Resource usage of ARGUS

IV. Takeaways and Future Work

ARGUS demonstrates that combining kernel-level eBPF telemetry with targeted Java and native profiling can effectively reconstruct comprehensive provenance graphs for multi-stage Android attacks. The core innovation lies in capturing Android-specific semantics, particularly structured Binder caller-callee relationships, alongside kernel-grounded evidence such as process, file, and network events. By integrating these data streams, ARGUS establishes a unified provenance graph that facilitates forensic backtracking and the reconstruction of complex attack narratives. Future work will focus on conducting extensive evaluations against real-world malware families and APT campaigns, expanding the coverage of sensitive Java and native hooks, and further analyzing how provenance graphs can drive practical investigation workflows on resource-constrained devices.

References

[1] Frida, "Frida dynamic instrumentation toolkit."

[2] J. Li, S. Chen, C. Wu, Y. Zhang, and L. Fan, "Foredroid: Scenario-aware analysis for android malware detection and explanation," in *CCS'25*.

[3] S. K. Smmarwar, G. P. Gupta, and S. Kumar, "Android malware detection and identification frameworks by leveraging the machine and deep learning techniques: A comprehensive review," *Telematics and Informatics Reports*, 2024.

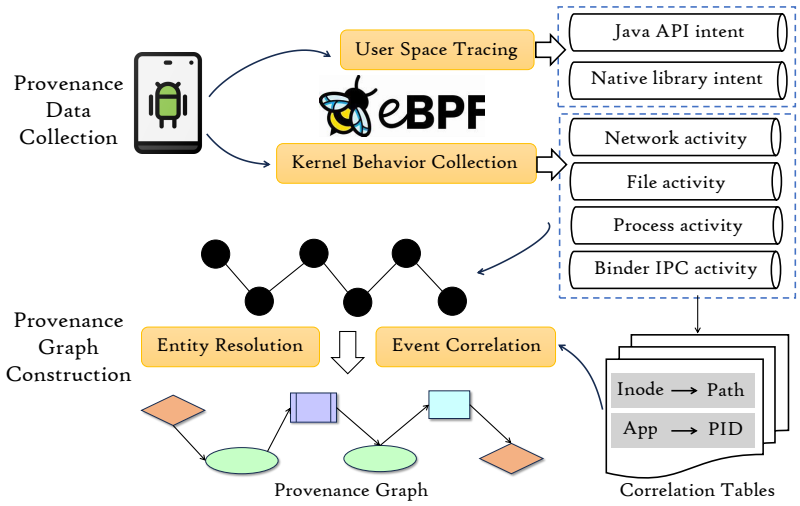
Reconstructing the Provenance of Android

Mingxiang Shi†, Xiangmin Shen ‡, Yuqiao Gu†, Zhipeng Chen †,
Lingzhi Wang‡, Yi Jiang†, Zhenyuan Li †
† Zhejiang University, ‡ Hofstra University, ‡ Northwestern University

Background & Insight

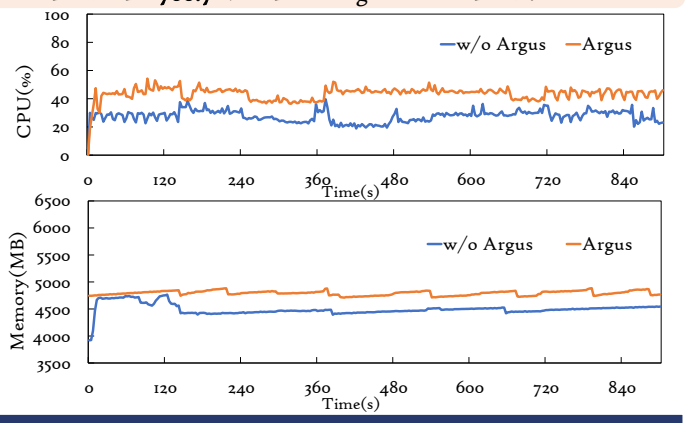
- **Problem**: Android incidents are hard to reconstruct end to end because attack steps can span Java APIs, Native libraries, Binder IPC, and kernel visible effects such as file and network operations.
- **Challenge**: Many sensitive actions run inside privileged system services on behalf of apps; syscall traces do not show the invoked Binder service interface or which service handled the request, so downstream effects are hard to attribute.
- **Limitation of existing tools**: Kernel-only tracing misses app intent and IPC semantics, while user-space-only hooks miss privileged and non-app behavior and are constrained by mobile overhead.

System Architecture & Methodology

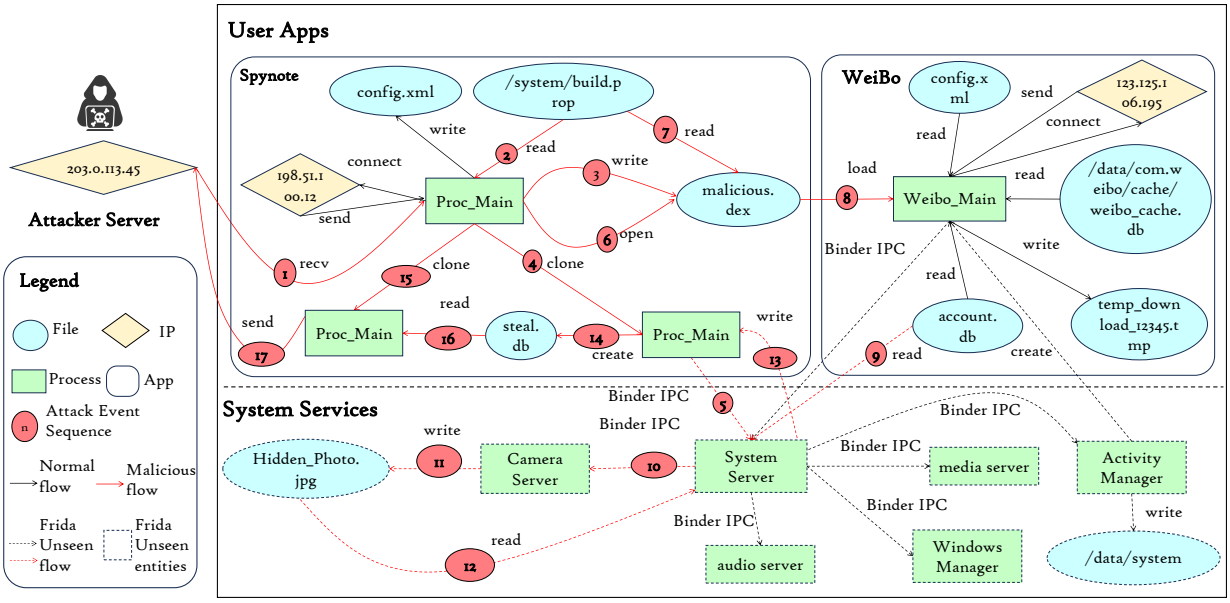


Overhead Measurement

- **CPU:** additional overhead **15.9%** in the low-load scenario and **12.5%** in the high-load scenario
- **Memory:** additional footprint of **297.8** MB in the low-load scenario and **766.7** MB in the high-load scenario.



Case Study: Multi-stage Android Attack



Attack Steps

Step 1: Spynote receives C2 commands, reads /system/build.prop, and downloads malicious.dex.

Step 2: The attacker escalates privileges and takes control of System_Server.

Step 3: The attacker uses System_Server to access protected data and trigger camera_server, storing outputs in steal.db.

Step 4: The attacker exfiltrates steal.db to remote infrastructure.

Takeaways

- **End-to-end evidence, not single-layer logs**: Build incident traces that connect app intent, IPC transitions, and kernel-visible effects to explain multi-step Android incidents.
- **Service-centric attribution**: Correct interpretation requires identifying which system service and interface executed a sensitive action, not just observing syscalls.
- **Lightweighted cross-layer monitoring**: Always-on investigation must recover essential semantics across app, services, and kernel while staying within mobile overhead constraints.