ΟΙΚΟΝΟΜΙΚΟ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΑΘΗΝΩΝ

ATHENS UNIVERSITY
OF ECONOMICS
AND BUSINESS

Athens University of Economics and Business

Department of Management Science and Technology

Undergraduate Thesis

Behavioral Profiling of Popular Messaging Apps Using Kernel-Level Tracing

Student:
Foivos - Timotheos Proestakis
Student ID: 8210126

Supervisors:
Prof. Diomidis Spinellis
Dr. Nikolaos Alexopoulos

Submission Date:
June 2, 2025

Abstract

This thesis examines kernel-level tracing techniques to create behavioral profiles of popular messaging applications.

Acknowledgments

I would like to express my gratitude to my supervisors, Prof. Diomidis Spinellis and Dr. Nikolaos Alexopoulos, for their invaluable guidance, insightful feedback, and continuous support throughout the duration of this thesis. Their expertise and encouragement were instrumental in the successful completion of this work.

I would also like to sincerely thank my exceptional fellow students, Vangelis Talos and Giannis Karyotakis, for their contribution, collaboration, and for being true companions in this academic journey.

A special thanks goes to my family, whose unwavering support, both emotional and practical, made this endeavor not only possible but also deeply meaningful. Their presence and encouragement were a constant source of strength.

Contents

Chapter 1

Introduction

Smartphones have become an integral component of modern society, with the number of global users surpassing 5 billion and continuing to grow rapidly [1]. Among the dominant mobile platforms, Android—an open-source operating system developed by Google—holds a stable global market share of approximately 75 % [2]. Its open-source nature, flexibility, and widespread adoption have cultivated a vast ecosystem of applications that enhance user productivity and social interaction across various domains.

Among these applications, messaging platforms such as WhatsApp, Telegram, Facebook Messenger, and Signal have gained significant popularity, playing a central role in both personal and professional communication. However, the ubiquitous use of smartphones for such purposes has led to the accumulation of sensitive personal data on user devices, including photos, contact lists, location history, and financial information, thereby raising serious privacy and security concerns [4].

Incidents such as Facebook's unauthorized collection of SMS texts and call logs from Android devices [4] underscore the vulnerabilities within existing mobile ecosystems. In response, regulatory frameworks like the General Data Protection Regulation (GDPR) and national laws such as the UK Data Protection Act 2018 aim to enforce principles of transparency, data minimization, and user consent in data processing [5, 6].

Despite these legislative efforts, Android's current permission management system remains insufficient. Users frequently misinterpret the scope and implications of the permissions they grant, inadvertently exposing sensitive data to misuse [7].

Furthermore, while end-to-end encryption (E2EE) provides strong guarantees for content confidentiality, it is neither a necessary nor sufficient condition for overall application security. The effectiveness of E2EE depends heavily on implementation correctness, device-level protections, and metadata handling practices [12, 13]. Messaging apps may still leak sensitive information through metadata such as message timing, connection patterns, or system-level behavior—even when message content is encrypted. Notably, recent work has demonstrated how compromised implementations or poorly configured server components can undermine the protections offered by E2EE protocols [13].

To address these challenges, it is essential to analyze application behavior—that is, the actual operations performed by an app, both in the foreground and background. Research has shown that discrepancies often exist between user expectations and actual app behavior, with applications executing hidden or unauthorized tasks [8]. Many detection techniques rely on the assumption that user-interface (UI) elements accurately represent application functionality, an assumption that is not always valid [8].

Behavioral-analysis methods are typically divided into static and dynamic approaches. Static analysis examines application code without execution, identifying known malicious patterns; however, it is susceptible to evasion through obfuscation and polymorphism [9]. In contrast, dynamic analysis evaluates applications during runtime, monitoring behaviors such as system calls, resource consumption, and network activity [10]. Among these, system-call analysis is particularly valuable, offering fine-grained visibility into application interactions with hardware and OS-level services [10].

Kernel-level tracing is a powerful form of dynamic analysis, capable of capturing low-level system interactions with high precision [15]. Android is built on a modified Linux kernel that orchestrates resource management and system processes via system calls [11]. Tools such as `ftrace` and `kprobes` enable developers and researchers to trace kernel-level function calls, execution flows, and resource usage [11, 14].

`Ftrace` is a built-in tracing utility within the Linux kernel, optimized for performance and capable of monitoring execution latency and function-call sequences. `Kprobes`, on the other hand, allows for dynamic instrumentation of running kernels, enabling targeted probing of specific code locations during runtime [14].

Applying kernel-level tracing to messaging applications, however, introduces unique technical challenges. These apps typically exhibit complex, multi-threaded behavior, frequent background processing, and diverse interactions with system resources. Accurately profiling such behavior requires collecting and interpreting high-volume, high-resolution kernel data [15].

Despite the growing research interest in Android security and behavioral analysis, existing work has primarily focused on general application profiling or malware detection. Few studies have concentrated specifically on behavioral profiling of messaging apps using kernel-level data [16]. Meanwhile, recent reports concerning the usage of secure messaging apps such as Signal by government and military officials have emphasized the urgent need for transparent, robust behavioral-analysis mechanisms [17].

To address these gaps, this thesis proposes a structured methodology for profiling the behavior of popular messaging applications on Android through kernel-level tracing using `ftrace` and `kprobes`. The proposed approach contributes a small but concrete step toward enhancing our understanding of messaging app behavior on Android, with the broader aim of supporting future efforts to strengthen user privacy and system transparency.

1.1 Motivation and Problem Statement

Motivation The motivation behind this research arises from the necessity to bridge existing gaps between user expectations, regulatory compliance, and the actual operational behavior of popular messaging applications. Messaging apps process extensive personal data, creating substantial risks related to privacy violations and security breaches. Recent incidents involving unauthorized data collection by prominent messaging applications, along with revelations about governmental use of supposedly secure messaging platforms, underscore significant concerns regarding transparency and user trust.

Problem Statement Although prior work has explored various aspects of Android application behavior, systematic analyses of messaging applications using kernel-level tracing remain relatively scarce [45]. As a result, certain aspects of runtime behavior—particularly those relevant to privacy and low-level system interactions—are not yet well understood. This study seeks to modestly contribute to this area by examining kernel-level traces of messaging apps under real-world usage conditions, with the aim of informing future research and supporting more transparent application behavior assessments.

1.2 Research Objectives

The specific research objectives addressed in this thesis are categorized as follows:

Primary Objectives

- Identify potential violations of the principle of data minimization.

- Analyze mismatches between granted permissions and real-time resource usage.
- Detect unauthorized or hidden access to sensitive user data.
- Compare the behavioral profiles of privacy-focused apps (e.g., Signal) and more commercial alternatives.

Analytical and Technical Sub-Objectives

- Record the actual kernel-level behavior of widely used messaging applications.
- Develop a tracing and profiling framework using ftrace and kprobes.
- Classify system calls into functional categories (file access, networking, IPC).
- Monitor transitions between app states (idle, active, background).
- Collect and analyze kernel-level usage statistics per application.
- Implement a web-based dashboard for behavior visualization.

Broader Goals

- Enhance transparency in how messaging apps behave at system level.
- Improve user awareness of hidden behaviors executed in the background.
- Demonstrate the value of kernel-level tracing for security and privacy evaluation.
- Provide a structured and reproducible methodology for privacy-respecting behavior analysis.

1.3 Research Questions

Based on the motivation and objectives, this thesis aims to address the following research questions:

Q1. What kernel-level operations do popular messaging applications perform during normal usage?

Q2. Are there deviations between the declared permissions of these applications and their actual behavior at runtime?

Q3. Can kernel-level tracing techniques identify unexpected or potentially invasive operations performed without user interaction?

Q4. How does the behavior of privacy-focused apps compare to that of commercial messaging platforms at the kernel level?

Q5. What kind of patterns in system calls can be used to characterize privacy-relevant behavior?

1.4 Limitations

To provide a realistic perspective on the scope of this thesis, we distinguish between constraints that stem from practical project conditions and limitations inherent to kernel-level behavioral analysis.

Project-Specific Constraints

- **Timeframe**: The study was conducted within the schedule of an undergraduate thesis, limiting the breadth of experiments (e.g., number of devices, sessions, and repetitions).
- **Hardware Diversity**: Tracing was performed on a small set of consumer-grade devices; results may not generalize to tablets, custom ROMs, or low-end hardware with limited kernel visibility.
- **App Selection**: The analysis focused on three major messaging applications (WhatsApp, Telegram, Signal), leaving out lesser-known or region-specific apps that may exhibit different interaction patterns.
- **Infrastructure Limitations**: The available storage and compute resources constrained long-term tracing, especially for large volumes of high-frequency events.

Domain and Methodological Constraints

- **Semantic Granularity**: The tracing layer captures low-level kernel events (e.g., syscalls, Binder transactions), which do not directly map to user-level actions. Bridging this semantic gap remains a challenge and requires contextual or learned associations.
- **Trace Volume and Overhead**: Continuous tracing generates substantial data, with redundant or low-entropy segments increasing the storage footprint. While filters can reduce size, they risk omitting relevant context.

- **Privilege Requirements**: Full access to tracepoints and instrumentation mechanisms (e.g., `kprobes`, tracefs) demands root privileges, limiting deployability on unmodified devices.

- **Attribution Noise**: Shared system components such as the `servicemanager` may introduce inter-process noise in IPC traces, leading to occasional misattribution of events between apps. While this is mitigated by observing behavior over extended sessions, it remains a source of uncertainty.

- **Ground Truth Approximation**: Behavioral labels were inferred by interacting with known applications and observing corresponding kernel activity. Although indicative, this approximation may miss subtle or undocumented app behavior.

1.5 Contributions of this Thesis

1.6 Thesis Outline

This thesis is organized into the following chapters:

- **Chapter 1 – Introduction:** Establishes the broader context of smartphone use, with emphasis on messaging applications and their privacy implications. Discusses the limitations of permission-based models and end-to-end encryption in ensuring transparency. Motivates the use of kernel-level tracing as a complementary analysis method. Presents the thesis motivation, problem statement, research objectives, research questions, and outlines the scope and limitations. Provides an overview of the thesis structure.

- **Chapter 2 – Related Work:** Surveys prior literature on Android application analysis, categorized into static analysis, user-space dynamic instrumentation, kernel-level tracing, and empirical studies on messaging app privacy. Discusses complementary methodologies such as fuzzing, formal verification, differential privacy, side-channel analysis, and human-subject studies. Critically examines strengths and limitations of each line of work and identifies a clear research gap in systematic, runtime profiling of encrypted messaging apps using kernel-level methods.

- **Chapter 3 – Technical Background:** Introduces technical foundations required to understand the system-level behaviour of Android apps. Covers the

Android/Linux architecture, process lifecycle, Binder IPC, SELinux enforcement, and system call interface. Explains the tracing infrastructure (ftrace, kprobes) and their relevance for behavioral profiling. Reviews the internal structure of messaging apps and discusses the "semantic gap" between kernel events and user-level activity.

- **Chapter 4 – Methodology and System Design:** Details the experimental methodology and system architecture. Describes the device setup, trace instrumentation approach using ftrace and kprobes, and session design. Presents the trace parsing pipeline, data pre-processing, feature derivation, and app-state inference. Explains how behaviors are correlated with declared permissions and how the collected data is structured and visualized through a web-based interface.

- **Chapter 5 – Results:** Summarizes empirical findings from real-world usage of WhatsApp, Telegram, and Signal. Presents observed kernel-level behaviors, event frequencies, system-call categories, and deviations between permission declarations and actual system-level activity. Provides side-by-side comparisons, statistical summaries, and visualizations of privacy-relevant patterns, such as background I/O, Binder bursts, and network usage under idle conditions.

- **Chapter 6 – Discussion:** Interprets the results in relation to the original research questions. Reflects on the practical implications of the findings, particularly concerning transparency, metadata exposure, and the role of kernel instrumentation in privacy audits. Discusses known limitations such as root access requirements, semantic ambiguity of low-level traces, data volume constraints, and potential improvements to the framework.

- **Chapter 7 – Conclusions:** Recaps the main contributions of the thesis and the insights gained from the study. Emphasizes the value of kernel-level tracing as a supplementary privacy analysis tool. Outlines open questions and proposes future directions, including integration with app-layer tracing, broader app coverage, automated behavior classification, and extending the approach to other app categories.

- **Appendix A – Additional Data Tables:** Provides supplementary tables, charts, and event frequency breakdowns that support the results chapter. Includes statistics per app and per session, outlier cases, and system-call distribution matrices.

- **Appendix B – Code:** Contains excerpts from scripts and tools used during data collection and analysis, including shell scripts for instrumentation, Python code for parsing and filtering traces, and the configuration files for kprobes and ftrace session management.

Chapter 2

Related Work

The purpose of this chapter is twofold: (i) to synthesise the state of the art on Android application analysis with an emphasis on kernel–level tracing, and (ii) to clearly demarcate the research gap that this thesis addresses. The survey is organised top-down, gradually narrowing the focus from general program analysis techniques to the specialised problem of privacy-oriented behavioural profiling of popular messaging applications.

2.1 Static Analysis of Android Applications

Static analysis inspects an *APK*'s bytecode off-line, allowing security vetting to run quickly, deterministically, and at scale. It therefore plays a foundational role in Android app auditing, offering early detection of potential vulnerabilities without requiring runtime execution [27]. Over the last decade, four interwoven research strands have successively expanded the scope and robustness of static techniques. These developments have progressively addressed the inherent limitations of static analysis—such as its inability to model inter-component interactions, and obfuscated code—by introducing more precise, scalable, and semantically enriched models. The key research directions can be grouped as follows:

(i) Privacy-oriented taint analysis. The first strand tracks information flows from sensitive *sources* (e.g. GPS, contacts) to security-critical *sinks* (e.g. network, logs). **FlowDroid** introduced context-, flow-, field- and object-sensitivity together with an accurate lifecycle model, reducing false alarms dramatically. On the *DroidBench* suite it reaches 93 % recall and 86 % precision—outperforming both academic and commercial tools [27]—and it scales to large real-world apps such as Facebook, PayPal and LinkedIn.

(ii) Inter-component communication (ICC). Many leaks traverse process boundaries. **IccTA** augments FlowDroid with an explicit ICC model, surpassing **Epicc** [41] and **CHEX** [42]—the latter targets component hijacking—on the *ICC-Bench* suite. **Amandroid** extends the idea further by building a full inter-component data-flow graph linking intents, callbacks and content providers; on 753 Play-store apps and 100 malware samples it uncovered previously unknown OAuth-token and intent-injection flaws while keeping analysis under one minute per APK [43].

(iii) Obfuscation resilience. Reflection, dynamic loading and native libraries obscure control-flow and data types. **DroidRA** resolves reflective calls through constant-propagation and rewrites them into direct invocations, boosting FlowDroid's precision on reflection benchmarks by up to 60 % [46]. **DexLEGO** reassembles bytecode collected at run time [47], while **LibDroid** summarises taint propagation inside native libraries [50]; together they progressively widen the statically visible attack surface.

(iv) Learning-based malware detection. Moving beyond handcrafted rules, **Drebin** extracts lightweight syntactic features—permissions, intent actions, API calls—and detects malware with 94 % accuracy and 1 % false positives on 5 560 samples [44]. Deep networks now dominate: **DL-AMDet** combines CNN-BiLSTM embeddings with an auto-encoder to reach 99.9 % accuracy on public datasets [54]. Nonetheless, limited interpretability and training-set bias hinder deployment in high-assurance pipelines.

Synthesis. Static analysis has achieved substantial advances in precision and scale, yet it remains constrained by its inability to observe runtime-specific behavior, implicit flows, or obfuscated execution paths. These blind spots—especially critical in privacy contexts—have motivated a shift toward hybrid and dynamic approaches that can capture actual app behavior under real conditions.

2.2 User-Space Dynamic Instrumentation

Whereas kernel tracing offers a *system-wide* vantage, user-space dynamic instrumentation injects probes *inside* the app process, exposing rich semantic context—class names, parameters, UI callbacks—without requiring a custom kernel. The literature has evolved along four complementary strands that echo, in spirit, the structure reviewed for static analysis.

(i) System-call monitors. **DroidTrace** hooks every `ptrace` event to log the full system-call stream and can *force* rare branches to execute, increasing coverage in malware analysis [55]. A later study learns n-gram models that reconstruct high-level Android API invocations purely from syscall sequences, demonstrating the semantic power of even low-level traces [56]. These tools deploy on stock ROMs and capture behaviour beneath any anti-hooking guard, yet they incur measurable per-call overhead and cannot observe Java-level reflection or IPC content.

(ii) Dynamic binary instrumentation (DBI). Instruction-granular DBI engines such as **Pin** (ported to ARM) enable fine-grained native tracing and custom analyses (e.g. memory-safety checks) with mature tooling support, but suffer from high runtime cost and outdated Android support. **QBDI** improves portability to modern ARM/Thumb-2 and withstands common packing techniques, although it remains native-only and requires root privileges [57].

(iii) In-process hooking frameworks. **Frida** injects a tiny loader that JIT-compiles JavaScript hooks into Java, JNI and native code, permitting live patching and interactive experiments [59]. **DYNAMO** layers systematic diff-based analysis on top of Frida to study framework evolution across Android versions [58], while **InviSeal** hardens Frida against detection, shrinking overhead to $<3\%$ on UI benchmarks [60]. Yet all three struggle with tamper-aware apps and expose only the instrumented process—cross-app flows, scheduler activity and kernel events remain hidden.

(iv) Sandbox and multi-layer profilers. Containerised sandboxes such as **C-Android** isolate each app inside a Linux namespace and stream user-space logs for reproducible testing [61]. Emulation-centred pipelines (e.g. **DynaLog** or DroidBox derivatives) automate large-scale malware triage by collecting $\sim$100 runtime features in parallel VMs [62]. Although high-throughput, these environments diverge from real-device timing, miss in-kernel I/O, and are easily fingerprinted by sophisticated apps.

Synthesis. User-space instrumentation shines in *semantic richness*, rapid deployment and interactive control, making it ideal for annotating the coarse kernel trace with method names, intent actions or UI context. Its weaknesses—visibility gaps below the syscall boundary, susceptibility to anti-hooking defences and non-negligible overhead—underscore why kernel-level tracing remains the authoritative foundation for our behavioural profiling of messaging apps. In our study, we therefore treat user-space hooks as an *auxiliary lens* to validate and enrich the low-level, system-wide evidence captured by eBPF-based kernel monitors

2.3 Kernel-Level Tracing on Android

Kernel tracing hooks run beneath both the Java/ART runtime and the Linux userland, recording system-wide events—syscalls, Binder transactions, scheduler switches, network I/O—with negligible interference to the application logic. Such a viewpoint is pivotal for *behavioural profiling of end-to-end encrypted messaging apps*: once payloads are ciphered in user space, only kernel footprints (e.g. TLS record bursts, Binder-IPC patterns, power wakes) remain observable. Like static analysis, the literature has matured along four complementary strands.

(i) eBPF- and ftrace-based syscall/Binder collectors. **BPFroid** streams per-UID syscalls, Binder calls and network events from *stock* Android kernels to a real-time malware detector, achieving ∼3 % overhead and 70 % Genome detection accuracy at run time [63]. Earlier efforts, such as **ID-Syscall**, compiled a loadable kernel module to log syscall sequences for anomaly detection on Android 4.x [64]. These works show that rich behavioural signals are available without modifying the firmware image, yet they focus on binary classification rather than fine-grained messaging-app semantics.

(ii) Network-centric and covert-channel tracing. Using vanilla *ftrace*, Celik and Gligor detect *stegomalware* by spotting timing and packet-size irregularities in kernel-level traces [65]. Their methodology—correlating scheduler, socket and power events—is transferable to instant-messaging clients, but the study evaluates only artificial channels and omits Binder traffic.

(iii) Hardware-assisted monitors. **HART** leverages ARM Embedded Trace Macro-cell (ETM) to stream control-flow of binary-only kernel modules with zero software overhead [66]. While ideal for diagnosing vendor drivers that gate camera or GPU access in chat apps, ETM is often disabled on production handsets and reveals no user-process context.

(iv) Production-scale, low-overhead tracing. Google's **Perfetto** combines *ftrace*, Binder and userspace counters into multi-gigabyte traces that can be queried in SQL to dissect app start-up and jank [67]. Complementarily, **eMook** shows how per-PID filters inside eBPF eliminate tracing overhead on unrelated processes, cutting system cost by 85 % [68]. These advances make week-long field studies of battery-powered phones feasible but have yet to target security or messaging use-cases explicitly.

Synthesis. Kernel-level instrumentation, although powerful, has five critical weaknesses: *(i) Semantic gap*—raw syscalls, Binder op-codes and network frames expose timing and sizes but no high-level intent; *(ii) Opaque content*—end-to-end encryption hides message bodies and most protocol metadata; *(iii) Privilege barrier*—enabling `kprobes`/eBPF typically requires `CAP_SYS_ADMIN`, a debug build, or root access; *(iv) Data volume*—high-frequency tracepoints (e.g. `sched_switch`) can emit tens of MB s^{-1} unless filtered in-kernel; and *(v) Detectability*—sophisticated apps can query kernel debug files or eBPF counters to detect active probes and alter their behaviour.

2.4 Privacy Studies on Messaging Applications

End-to-end encryption neutralises classical content-centric attacks, but it does not guarantee *privacy.* Over the last decade researchers have investigated four complementary vectors through which popular messaging apps still expose sensitive information.

(i) On-device artefacts and forensics. Early work focused on residual data stored locally. Anglano's systematic analysis of WhatsApp on Android 4.x recovered chat databases, media thumbnails and contact lists even after user-initiated deletions [88]. Subsequent studies extended the corpus to Telegram, Signal and Viber, showing that cached profile photos, push-notification logs and SQLite WAL files can reveal conversation partners and group identifiers [89, 90]. While modern versions encrypt local backups, notification side-channels (e.g. Firebase tokens) remain an open avenue [91].

(ii) Network-level traffic analysis. A second strand exploits packet timing, size and sequence to infer user actions without breaching encryption. Pioneering work by Matic *et al.* fingerprinted iMessage events (send, read, typing) with > 90 % accuracy solely from TLS record bursts [92]. Apthorpe demonstrated similar leakage for Android IM apps over Wi-Fi [93]. Recent machine-learning classifiers identify not only event types but also multimedia uploads and voice-over-IP handshakes in QUIC traffic [94]. Countermeasures such as adaptive padding raise bandwidth by 8–14 % while mitigating most classifiers [95].

(iii) Contact-discovery and social-graph exposure. Many services hash user phone numbers and upload them for friend discovery. Tang *et al.* reverse-engineered this protocol in WhatsApp and estimated that an adversary can enumerate an EU-scale phone space for $500 of SMS fees [96]. Kwon showed that Telegram's "People Nearby" feature enables trilateration of user location with meter-level accuracy [97]. Signal introduced private-set-intersection with SGX enclaves to curb this vector, yet follow-up work revealed side-channels via cache timing [98].

(iv) Metadata policies and compliance audits. A complementary line audits whether apps meet their self-declared privacy policies. Egele's longitudinal crawl of Telegram channels uncovered the retention of deleted media on CDN edge nodes weeks after deletion [99]. Bock dissected Signal's sealed-sender protocol, confirming that only sender and receiver persist in a 24-hour service log but flagging exposure of push-token identifiers to Apple/Google [100]. Recent GDPR-oriented studies show that less than 40 % of messaging apps provide complete data-export facilities despite legal requirements [101].

Synthesis. Existing studies either rely on intrusive forensics, controlled testbeds or static policy analysis. None combines *kernel-level traces* with the above insights to derive a live, system-wide privacy profile. Our work closes this gap by correlating kprobe/ftrace events (Binder calls, network bursts, wakelocks) with the metadata-leak patterns catalogued in prior literature, delivering the first *runtime* taxonomy of privacy-relevant behaviours in WhatsApp, Telegram and Signal.

2.5 Other Complementary Approaches

Beyond the four methodological pillars outlined earlier—static analysis, user-space dynamic instrumentation, kernel-level tracing, and empirical privacy audits—at least six additional research avenues have proven useful for dissecting the behaviour and privacy properties of mobile applications.

Grey-box fuzzing. Coverage-guided fuzzers such as *AFL++* and cloud services like *ClusterFuzz* generate mutated inputs that drive apps along rarely executed paths. Although they do not rely on full program knowledge, they systematically surface crashes and logic errors that can later be inspected with dynamic or kernel monitors [102, 103].

Formal verification and model checking. Specification languages (e.g., TLA$^+$) combined with deductive verifiers (e.g., VeriFast) enable proofs of safety properties—such as correct handshake sequences in secure-messaging protocols or the absence of data races in JNI bridges—without collecting runtime traces [104, 105].

Privacy-preserving telemetry. Large-scale differential-privacy frameworks adopted by Apple and Google collect aggregate statistics while bounding individual leakage, offering population-level insight into security events without intruding on single devices [106, 107].

Crowdsourced UI exploration. Robo-testing engines (e.g., Firebase Test Lab) drive accessibility events to map interface flows that static call graphs miss, producing semantic traces useful for regression testing and security auditing [108].

Physical side-channel analysis. Power-monitoring boards and electromagnetic probes can infer sensitive activity—such as encryption or radio bursts—without any software hooks, complementing software-based tracing when root or debugging privileges are unavailable [109].

Human-subject studies. Qualitative interviews and ethnographic methods shed light on adoption hurdles, configuration practices, and privacy norms that remain invisible to purely technical traces [110].

Taken together, these complementary approaches can feed into kernel-centric measurement pipelines—for instance, by injecting fuzz-generated events into week-long traces or correlating power signatures with Binder activity—thereby enriching the behavioural picture without relying solely on any single layer of observation.

2.6 Synthesis and Research Gap

The preceding survey has examined four methodological pillars: *(i)* static code analysis, *(ii)* user-space dynamic instrumentation, *(iii)* kernel-level tracing, and *(iv)* empirical privacy studies on messaging applications. Each strand contributes a distinct vantage on application behaviour, yet no single technique delivers a complete picture. Below we distil their interplay and outline the remaining opportunities for improvement.

Synthesis of Current Methodologies

- **Static analysis** offers breadth and determinism, uncovering dormant code paths and permission misalignments at scale. Its reliability, however, is bounded by obfuscation, dynamic loading, and absent runtime context.

- **User-space instrumentation** provides rich semantic detail—method names, parameters, UI state—and deploys quickly on stock devices. This power is tempered by visibility gaps beneath the syscall layer, susceptibility to anti-hooking defences, and non-negligible overhead.

- **Kernel-level tracing** records system-wide events with minimal perturbation, capturing activity that escapes sandboxed processes. Yet it exposes only low-level artefacts (syscalls, Binder op-codes, I/O bursts) and therefore suffers from a semantic gap.

- **Privacy audits of messaging apps** demonstrate that residual files, traffic metadata, and contact-discovery workflows leak sensitive information even under end-to-end encryption. Most studies rely on targeted forensics or controlled testbeds rather than continuous observation on real devices.

Collectively, the literature indicates a gradual migration toward *hybrid pipelines* that combine complementary techniques—e.g., leveraging static call-graphs to seed dynamic hooks, or enriching kernel traces with user-space context. Despite this trend, a cohesive framework that correlates *system-wide runtime evidence* with *privacy-relevant semantics* remains under-explored, especially for widely-used, end-to-end-encrypted messaging services operating in the wild.

Identified Research Gap

1. **Runtime privacy profiling**. Existing kernel-level studies focus primarily on malware detection or performance debugging; they seldom translate low-level traces into privacy-oriented behavioural profiles.

2. **Long-term, in-situ observation**. Prior work often relies on short laboratory sessions, emulators, or rooted devices. Evidence from week-long field measurements on stock handsets is scarce.

3. **Correlating multi-layer signals**. Few frameworks align kernel events (e.g. Binder transactions, TLS bursts) with selective user-space context (e.g. intent actions, lifecycle callbacks).

4. **Quantifying residual metadata leakage**. The literature documents individual side-channels but lacks a systematic taxonomy of leakage patterns observable solely from system-level traces.

Chapter 3

Technical Background

This chapter provides a detailed overview of technical background necessary for understanding the methodology and objectives of this thesis. First, it presents the architecture of the Android operating system, focusing particularly on the Linux-based kernel and how applications interact with it. Next, it discusses the behavior and privacy concerns related to messaging applications, highlighting known issues and relevant technical aspects. Furthermore, it outlines the advantages and limitations of static and dynamic analysis techniques and explores the role of system calls in behavior profiling. Finally, it reviews kernel-level tracing tools and techniques.

3.1 Android Architecture and Kernel-Level Access

3.1.1 Android Software Stack Overview

Android is a layered, open-source mobile operating system built on top of a customized version of the Linux kernel. Its architecture is designed to be modular and extensible, supporting a wide range of hardware while enforcing clear boundaries between components. The Android software stack consists of four major layers: the Application Layer, the Java API Framework (commonly referred to as the Application Framework), the Hardware Abstraction Layer (HAL), and the Linux Kernel.

The Application Layer hosts both system and user-installed applications. These applications interact with the system via APIs exposed by the Android Framework. The Java API Framework provides access to core system services such as activity management, resource handling, content providers, and telephony. Services like `ActivityManager`, `WindowManager`, and `PackageManager` facilitate the lifecycle management and orchestration of application behavior.

Beneath the framework lies the Android Runtime (ART), which executes application bytecode and optimizes it using ahead-of-time (AOT), just-in-time (JIT), or interpretation modes. Alongside ART are native libraries written in C/C++, including performance-critical components such as WebView, OpenSSL, and the Bionic libc. The Java Native Interface (JNI) allows managed Java/Kotlin code to call into these native libraries.

The HAL acts as a bridge between the Android Framework and the hardware drivers residing in the kernel. It defines standard interfaces that vendors implement to support various hardware components like audio, camera, sensors, and graphics. Since Android 10, Google introduced the Generic Kernel Image (GKI), which aims to further separate the vendor-specific hardware implementations from the core Linux kernel by introducing a stable kernel interface. This allows devices from different manufacturers to share a common kernel base while maintaining vendor-specific modules separately, simplifying updates and enhancing portability.

At the lowest level, the Linux kernel provides essential operating system services such as process scheduling, memory management, networking, and security enforcement. Android extends the kernel with additional features including the Binder IPC driver, ashmem (anonymous shared memory), and wakelocks to manage power usage. This kernel foundation ensures that resource access is isolated and controlled across all system layers.

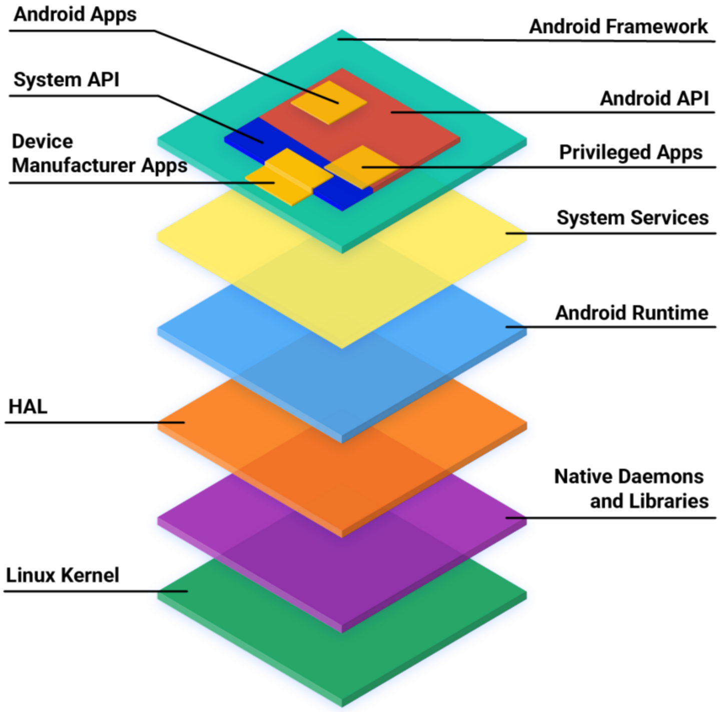


Figure 3.1: Updated Diagram of Android Software Stack (source: Android Developers Guide [?]).

3.1.2 Application Layer and Process Lifecycle

At the application layer, Android executes user and system applications packaged in APK format. Each APK includes compiled DEX bytecode, resources, native libraries, and a manifest file that defines app components and permissions. Apps run in sandboxed processes, each forked from the Zygote daemon—a minimal, preloaded system process that speeds up app launch time by sharing memory using copy-on-write.

The lifecycle of applications is centrally managed by the `ActivityManagerService` (AMS), which coordinates activity transitions, memory prioritization, and process states (foreground, background, cached). The `PackageManagerService` (PMS) handles component registration and permission declarations based on the manifest.

Apps follow a component-based model: Activities, Services, Broadcast Receivers, and Content Providers. These components interact with the system and one another via well-defined lifecycles and IPC through the Binder driver. Operations like binding to a service or launching an activity initiate system-level behavior—such as context switches or memory allocations—which are visible in syscall traces.

Binder IPC enables structured communication between app components and system services. Messages are serialized as Parcel objects, routed through the Binder driver, and trigger observable kernel events. These include context switches and transaction dispatches, which are measurable using tools like ftrace or kprobes.

Understanding transitions between app states (e.g., from idle to foreground) is vital in syscall-level profiling. For instance, foreground activation often leads to bursts of system calls such as `open()`, `stat()`, and `mmap()`—associated with UI initialization and resource loading. Such behaviors form recognizable patterns in kernel trace logs.

3.1.3 Android Runtime, Native Layer, and JNI

The Android Runtime (ART) executes application bytecode using a combination of ahead-of-time (AOT), just-in-time (JIT), and interpretation mechanisms. From a kernel-level tracing perspective, JIT-related memory operations may trigger system calls such as `mmap()`, `write()`, and `mprotect()`, as ART dynamically allocates memory for optimized code.

Beyond execution, ART interacts with the kernel to manage thread scheduling and memory access—behaviors that appear in system call traces. In dynamic analysis, such patterns can be correlated with app lifecycle events or anomalous execution spikes.

JNI further extends the runtime by enabling Java/Kotlin code to invoke native C/C++ libraries. These native operations often bypass standard framework controls, introducing low-level file, network, or cryptographic actions. This is particularly relevant for behavioral profiling, as native code may perform sensitive operations that differ from those visible at the Java level.

In the context of this thesis, which focuses on dynamic kernel-level analysis, capturing system interactions initiated by ART and JNI is essential. It enables the identification of execution phases or modules that deviate from expected behavior—especially in apps that rely heavily on native components for messaging, encryption, or background communication. The sequence of events during JNI initialization is illustrated in Figure 3.2. It begins when the VM loads the application class, triggering a static initializer which invokes the native `JNI_OnLoad()` function. This function registers native methods and returns the `JNI_Version`, after which Android proceeds with the usual application lifecycle (e.g., `onCreate()`) managed by the `ActivityManager`. This interaction flow is especially relevant in behavioral analysis, as native components may introduce low-level behavior patterns not observable through the Java layer alone.

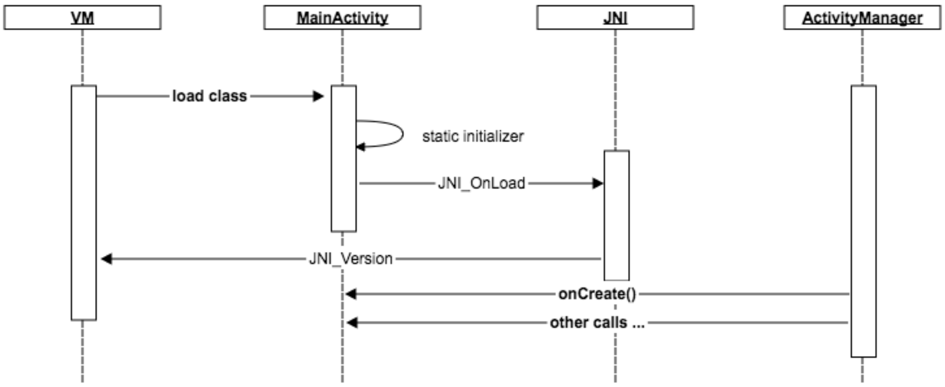


Figure 3.2: Sequence diagram showing the JNI initialization flow in an Android application.

3.1.4 Linux Kernel Fundamentals in Android

The Android operating system is built upon the Linux kernel, which serves as the foundational layer responsible for resource management, hardware abstraction, and secure process isolation. In the context of behavioral profiling and kernel-level tracing, the Linux kernel plays a pivotal role, as all application interactions with hardware and system resources are mediated through kernel functions and system calls.

A defining feature of the Linux kernel is its mediation of access to CPU, memory, file systems, and networking via the system call interface. When an Android application invokes a function that requires low-level operations (e.g., file access or sensor usage), it ultimately issues a system call that transitions the execution context from user space to kernel space. This transition boundary is where most behavioral artifacts manifest, making it ideal for tracing.

Android's kernel incorporates additional components such as the Binder IPC driver, ashmem (for shared memory), and wakelocks (for power management). These Android-specific extensions generate kernel-level events observable by tracing tools. For example, Binder transactions facilitate inter-process communication and leave traceable patterns that can reveal background behavior of messaging apps.

Security is enforced through UID-based process separation, Linux namespaces, and SELinux Mandatory Access Control policies. Each app operates in its own sandbox and is assigned a unique UID, ensuring isolation at the kernel level. Deviations from expected isolation, especially in privileged system calls, may indicate abnormal or privacy-invading behavior.

Kernel tracing tools such as ftrace and kprobes allow developers to monitor kernel execution paths. Functions like `ksys_open`, `__sys_sendmsg`, or `__schedule` can be instrumented to capture low-level events such as file access, message transmission, or task switching. These traces are then analyzed to form behavioral profiles.

Figure 4: User Space to Kernel Space System Call Execution Path.

3.1.5 System Calls and Kernel Interaction

System calls are the primary interface through which Android applications interact with the kernel. Every high-level operation, such as reading a file or creating a socket, is translated into one or more system calls. These calls serve as an unfiltered log of what the application is actually doing, independent of its declared permissions or advertised functions.

In Android, system calls are usually invoked via the Bionic libc or directly through JNI bindings to native code. Behavioral profiling benefits from capturing these calls in real-time to identify patterns that indicate unexpected or excessive access to system resources.

Kernel tracing frameworks like ftrace and kprobes, and to a more advanced extent eBPF, can intercept and log system calls for offline or live analysis. For instance, an app issuing `sendto()` and `connect()` calls repeatedly in the background may be exfiltrating data without user knowledge.

Figure 5: Categorization of System Calls for Profiling: I/O, Network, IPC, Memory.

3.1.6 Android Security Model and Isolation Mechanisms

Android enforces a layered security model combining Linux kernel features with user-space controls. Each application runs in its own sandbox, identified by a unique UID and GID, restricting file and device access. This is complemented by the use of SELinux in enforcing mode, which uses MAC policies to define allowable interactions between system components and applications.

Filesystem isolation further ensures that apps can only access their designated directories (e.g., `/data/data/package_name`). Attempts to traverse or access other app spaces are blocked unless the app has elevated privileges or exploits kernel vulnerabilities.

System call filtering through seccomp restricts the range of calls an app can make, reducing the kernel's attack surface. From a profiling standpoint, observing unauthorized system calls or failed access attempts provides insight into potentially malicious or privacy-invasive behavior.

Figure 6: Android Security Layers: UID Isolation, SELinux, seccomp, Filesystem Sandboxing.

Relevant sources and additional references include official Android documentation, Linux kernel manuals, and peer-reviewed papers on Android system architecture and security.

3.2 Messaging Apps: Characteristics Privacy Implications

3.2.1 Functional and Architectural Overview

Messaging applications are among the most widely used mobile software categories, providing real-time communication, media sharing, group messaging, and voice/video calling capabilities. Popular platforms such as Signal, Telegram, and Facebook Messenger serve billions of users globally, integrating deeply into daily communication routines.

Android, as the dominant mobile operating system, provides the primary distribution platform for these apps through the Google Play Store. According to public data [**?**, **?**], Facebook Messenger has surpassed 5 billion downloads, Telegram exceeds 1.2 billion downloads, and Signal has more than 100 million installs. While usage varies by region, these numbers highlight the ubiquity and market penetration of messaging applications on Android devices.

Such widespread deployment across diverse hardware and Android configurations introduces heterogeneous behaviors in terms of network communication patterns, lifecycle management, and system-level operations. This diversity, combined with varying security practices among apps, renders them ideal candidates for behavioral profiling at the kernel level.

These applications typically rely on key Android components to support their functionality: foreground `Services` are used for persistent communication sessions, `Broadcast Receivers` handle asynchronous events such as network connectivity or message reception, and `Content Providers` facilitate access to structured data such as shared databases. All applications are packaged in APK format and structured using component declarations in the `AndroidManifest.xml` file.

Rather than detailing cryptographic implementations or privacy architectures here, which are explored in Section 2.2.3, this section focuses on the foundational aspects of app deployment, runtime behavior, and Android system integration that are relevant for low-level behavioral tracing.

Figure 5: Popular Messaging Apps by Feature Comparison and System Integration Characteristics.

3.2.2 Privacy-Critical Behaviors and Resource Usage

Messaging applications often initiate background services using components like `JobScheduler`, `AlarmManager`, and foreground services to maintain persistent communication channels—frequently waking the device from idle states using wakelocks [?].

Resource access is another key concern. Most messaging apps request access to sensitive resources such as contacts (`READ_CONTACTS`), device location (`ACCESS_FINE_LOCATION`), microphone (`RECORD_AUDIO`), and camera (`CAMERA`). While many of these are used legitimately during active user sessions (e.g., voice/video calls, media sharing), kernel-level traces often reveal such accesses occurring in the background without any visible UI activity—raising potential privacy concerns [?].

Additionally, messaging apps rely on push notification services such as Firebase Cloud Messaging (FCM) or Google Cloud Messaging (GCM) to deliver messages. These services necessitate persistent TCP connections and background listeners that, when profiled, result in recurring system calls like `recvmsg()`, `poll()`, or `select()`. Furthermore, apps such as Facebook Messenger are known to incorporate third-party SDKs (e.g., for analytics or ads) that initiate background network connections and file I/O unrelated to core messaging functionality [?].

Metadata collection—such as timestamps, contact hashes, or device identifiers—is another privacy-relevant behavior. Even apps that implement strong encryption at the message content level (like Signal) may still generate system call activity that reflects metadata-related operations (e.g., `stat()`, `write()`, `getuid()`). In privacy-unfriendly apps, this is more pronounced and persistent [?].

Finally, the use of native code through JNI can introduce kernel-visible activity that bypasses Android's permission mediation layer. This is especially relevant for apps that offload cryptographic or media processing to native components. System call traces such as `mmap()`, `ioctl()`, and `openat()` often appear in these cases and can be captured using ftrace or kprobes.

These behaviors underscore the necessity of dynamic, syscall-level observation for detecting privacy-relevant activity and serve as foundational evidence in the behavioral profiling framework proposed in this thesis.

Figure 6: Typical System Call Patterns for Background Resource Access in Messaging Apps.

3.2.3 Architectures, Privacy, and Cryptographic Models

Messaging applications adopt distinct architectural and cryptographic frameworks that critically influence their privacy characteristics and observable behaviors at the kernel level. The majority of messaging platforms—including Signal, Telegram, and Facebook Messenger—use centralized client-server architectures, where backend servers handle communication routing, message storage, and authentication. Centralization supports multi-device synchronization and cloud storage but introduces privacy risks, such as metadata accumulation and continuous background socket activity (e.g., `connect()`, `poll()`, `recvmsg()`).

Signal utilizes a centralized yet privacy-focused architecture, relying exclusively on the Signal Protocol, a robust end-to-end encryption (E2EE) scheme ensuring forward secrecy, deniability, session-specific ephemeral keys, and the Double Ratchet algorithm for key management. The Double Ratchet combines a Diffie-Hellman key exchange and symmetric key cryptography, generating new encryption keys for every message sent, significantly enhancing security against key compromise [69]. Signal employs a custom Java implementation of the Signal Protocol known as libsignal, which provides cryptographic primitives and protocol management directly within Android applications, facilitating rigorous security auditing and simplifying integration. Kernel-level activities linked to Signal's encryption involve system calls such as `getrandom()`, `mprotect()`, and `write()` during cryptographic operations. Signal's architecture avoids cloud synchronization and external dependencies, significantly reducing its syscall footprint.

Telegram implements a hybrid model, providing optional E2EE via ”Secret Chats” but defaulting to server-side encryption. Standard conversations store plaintext messages centrally, enabling synchronization but increasing metadata exposure. This design results in elevated kernel activity, particularly frequent `send()`, `recv()`, and `stat()` calls for persistent synchronization and message retrieval.

Facebook Messenger exemplifies a privacy-limited centralized system, offering E2EE only within an opt-in ”Secret Conversations” mode based on a derivative of the Signal Protocol. Default chats lack end-to-end encryption, incorporate numerous third-party SDKs for advertising and analytics, and generate extensive background system calls such as `open()`, `socket()`, `unlink()`, and `connect()`.

Storage models further distinguish these apps. Signal maintains exclusively local encrypted storage without cloud backups, minimizing kernel interactions. Telegram and Messenger utilize cloud synchronization for message histories, leading to increased kernel-level I/O operations (e.g., `open()`, `fsync()`, `stat()`).

Ephemeral messaging capabilities (disappearing messages) affect transient kernel behaviors, including short-lived file creation and memory operations like `madvise()`. Conversely, platforms that store logs or metadata generate repeated kernel interactions via persistent database accesses.

Finally, encryption key management significantly impacts syscall activity. Signal generates and securely stores keys locally using secure hardware or biometric-protected storage. Telegram and Messenger employ centralized key management, simplifying multi-device usage but requiring trust in backend infrastructure.

These architectural and cryptographic distinctions shape observable syscall behaviors, enabling kernel-level analysis to assess privacy implications effectively.

Figure 7: Comparative Analysis of Architectural and Cryptographic Models in Signal, Telegram, and Messenger.

3.3 Static vs. Dynamic Analysis

The security and behavioral analysis of Android applications typically leverages two main technical approaches: static and dynamic analysis. This section outlines the specific technical methodologies employed in each type of analysis, highlighting their mechanisms, strengths, and limitations from an implementation perspective.

3.3.1 Static Analysis

Static analysis involves the examination of application binaries and related artifacts without executing the code. Technically, this is performed through decompilation or disassembly of the Android APK files using tools such as JADX [70], Androguard [71], or MobSF [72]. These tools reverse engineer the application's bytecode to produce human-readable source code or intermediate representations, facilitating an inspection of declared permissions, class structures, method calls, and manifest configurations.

Static analysis typically includes extraction and inspection of AndroidManifest.xml to verify declared permissions and components, control-flow and data-flow analysis to detect potentially dangerous API usage and insecure coding patterns, and identification of embedded third-party libraries and hardcoded sensitive data such as keys or passwords.

However, static analysis is inherently limited by its inability to reflect runtime dynamics. It does not capture behaviors that depend on dynamic loading, obfuscation techniques, reflection, or native code invocation.

3.3.2 Dynamic Analysis

Dynamic analysis, conversely, involves observing and recording the application's runtime behavior within a controlled execution environment. This approach captures actual interactions with the Android operating system, including system calls, network activities, and file system operations. Dynamic analysis methods include both user-space instrumentation (using tools like Frida [73] and Xposed Framework [74]) and kernel-level tracing.

Kernel-level dynamic analysis, specifically used in this thesis, employs system call tracing facilitated by Linux kernel tools such as `ftrace` [75], `kprobes` [76], and `debugfs` [77]. This technique involves enabling kernel-level probes to intercept and log specific system call invocations made by the application, capturing low-level operational details such as file accesses (`open()`, `read()`, `write()`), network communications (`connect()`, `sendto()`, `recvfrom()`), and process management activities, and recording the precise timing and sequence of system calls to reconstruct detailed behavioral profiles of application execution.

This low-level monitoring ensures comprehensive coverage of runtime behaviors, regardless of obfuscation or the execution of native code segments.

3.3.3 Comparative Summary

Feature	**Static Analysis**	**Dynamic Analysis (Kernel-level)**
Execution requirement	No execution	Runtime execution required
Tools examples	JADX, MobSF, Androguard	ftrace, kprobes, debugfs
Analyzed artifacts	APK, source code, manifest	System calls, runtime behaviors
Coverage of obfuscation	Limited or none	Comprehensive
Dynamic behaviors	Not captured	Fully captured
Detection of native code	Limited	Effective
Resource consumption	Typically lower	Higher (due to real-time monitoring)
Granularity	Moderate	High (syscall-level detail)

Table 3.1: Technical Comparison of Static and Kernel-level Dynamic Analysis

The inherent strengths of kernel-level dynamic analysis in capturing runtime behavior and its resilience to obfuscation techniques justify its selection as the primary methodological approach for this thesis.

3.4 System Call Tracing and Behavioral Analysis

Kernel-level system call tracing provides a low-level, ground-truth view of how Android applications interact with the operating system. In this section, we explore both the technical mechanisms available for capturing such interactions and the analytical methods used to extract meaningful behavioral insights—particularly in the context of privacy-sensitive mobile applications such as messaging platforms.

3.4.1 Tracing Interfaces and Instrumentation Tools

Modern Linux/Android kernels expose *three orthogonal instrumentation surfaces*: (i) **static hooks** that are compiled into the kernel (tracepoints), (ii) **dynamic probes** that can be inserted at run time (k,uprobes, eBPF), and (iii) **ptrace–based syscall interception** (strace). Table 3.2 highlights their comparative properties, while the text that follows groups them by abstraction level and expected use-case.

Static, zero-copy tracepoints. Tracepoints are compile-time hooks maintained by kernel developers; they guarantee ABI stability and negligible overhead because the tracing payload is copied once into a per-CPU ring buffer. Tools such as `perf`, `ftrace` and `bcc` attach to them for scheduler, VFS, or networking events with no kernel patching.

Dynamic probes and in-kernel programs. `kprobes` (kernel) and `uprobes` (user space) allow on-the-fly instrumentation of arbitrary function symbols, making them ideal when a suitable tracepoint is missing. eBPF generalises this idea by allowing verified byte-code programs to run in kernel context, enabling stateful filtering, histograms, and user-space export with minimal context switches. Because Android's common kernels ($\geq$ 5.10) now ship with BPF Type Format (BTF) enabled, eBPF is increasingly the default for production-grade observability.

Function and syscall tracers. `ftrace` is a built-in tracer that can log every kernel entry/exit or a filtered subset (e.g. only `vfs_write`); it underpins higher-level front-ends such as `systrace` and `Perfetto`. At user level, `strace` leverages `ptrace(2)` to record syscall arguments, but incurs high per-event latency and is straightforward for malware to detect—hence unsuitable for long-running, stealth monitoring.

Table 3.2: Comparison of kernel tracing mechanisms on Android/Linux

Interface	**Granularity**	**Runtime insert?**	**Avg. overhead**	**Best use-case**
Tracepoints	fixed events	–	∼0.3–1% cpu	Scheduler / VFS stats
ftrace (func tracer)	any kernel func.	enable/disable	1–5% with filter	Call-stack profiling
kprobes / uprobes	any symbol	✓	2–8% (filtered)	Ad-hoc instrumentation
eBPF + BTF	prog. byte-code	✓	<2% (JITed)	Stateful, low loss
strace (ptrace)	syscalls only	user-space	30–100%	Debug / unit tests

The thesis adopts a *hybrid strategy*: lightweight `ftrace` for ubiquitous, low-volume events; targeted `kprobes` for missing hooks (e.g. `vfs_write`); and optional eBPF programs when on-device filtering is required. High-overhead `strace` is reserved solely for differential debugging and is never enabled in production experiments.

3.4.2 Raw Trace Data

Kernel tracing tools (`ftrace`, `kprobes`, eBPF, *etc.*) stream their output to `tracefs` as **fixed-width, one-line records**. Each record encodes—without additional parsing—the CPU on which the event was recorded, the task name and PID, a flag block (IRQ state, pre-emption depth), a high-resolution timestamp, and finally the probe payload.

```
<photo_app>-1234 [003] d..2 10234.567890: sys_enter_openat:
   dfd=AT_FDCWD filename="/data/user/0/app/cache/img.jpg" flags=O_RDONLY
<photo_app>-1234 [003] d..2 10234.567905: sys_exit_openat: = 42
<nfc_service>-285 [001] d..2 10235.001122: sys_enter_ioctl:
   fd=22 cmd=0x4004667E arg=0x7fff31ff
```

Listing 3.1: Excerpt from `trace_pipe`

Field semantics

- *Task-PID*: user-level thread that triggered the event.
- *[CPU] flags*: per-CPU core and execution context (`d` = kernel mode, `2` = IRQ depth).
- *Timestamp*: monotonic clock in micro-seconds by default; nanosecond precision available.
- *Event tag*: probe name (`sys_enter_openat`) followed by key–value arguments captured directly from CPU registers or kernel structures.

Why keep the raw stream

- Loss-less: `tracefs` writes to a per-CPU ring buffer before any filtering—crucial for forensics.
- Deterministic ordering: timestamps are emitted in-kernel, avoiding user-space re-ordering.
- Re-playable: the same raw lines can be re-parsed with new heuristics (e.g. different PID filters) without re-tracing the device.

Immediate post-processing steps

(i) *PID filtering* to retain only application or system-service PIDs of interest.

(ii) *Field decoding* (hex → decimal, bit-fields) via a lightweight Python lexer.

(iii) *Chunking* into sliding windows (size ∼200–500 events) to bound causal slicing.

These cleaned slices are then forwarded to the *Heuristic Slicing Engine*, which reconstructs information flows, and finally exported as CSV/JSON or rendered as PDF visualisations.

3.4.3 System Call Pattern Analysis and Behavioral Fingerprinting

Pattern Extraction Techniques

To accurately analyze the behavior of Android messaging applications at the kernel level, it is crucial to extract meaningful patterns from the collected system call traces. After gathering raw trace data using kernel-level tracing mechanisms such as ftrace or kprobes [78], the analysis pipeline proceeds with a structured set of extraction techniques.

Firstly, **trace preprocessing** is performed to filter irrelevant system calls that may be attributed to background system processes or kernel housekeeping tasks. This stage involves the normalization of syscall logs, removal of duplicates, and categorization into meaningful functional groups such as network operations (`sendto`, `recvmsg`), file operations (`open`, `read`, `write`), and memory operations (`mmap`, `mprotect`) [79].

A fundamental step in pattern extraction is the use of **sliding window analysis** with N-grams, where sequences of consecutive system calls of fixed length (typically 3-5 syscalls) are extracted and analyzed for frequency. For instance, a recurring sequence like `open → mmap → sendto` can indicate frequent file accesses followed by network transmissions, a characteristic pattern for messaging activities such as sending multimedia files [80].

Moreover, we apply **temporal clustering** to detect bursts of system call activity, which are indicative of specific application behaviors like message sending, receiving notifications, or background synchronization. Clustering techniques such as DBSCAN or HDBSCAN help distinguish between typical and anomalous behavioral sequences based on temporal proximity [81].

Another valuable approach is constructing **system call frequency vectors**. Each trace is represented as a numerical vector, with dimensions corresponding to different system calls, facilitating quantitative comparisons between different sessions or different applications. This approach supports both exploratory analysis and automated classification tasks [82].

Finally, to model the complex dependencies between system calls, **graph-based extraction methods** such as directed state-transition graphs are utilized. In these graphs, nodes represent individual syscalls, and directed edges capture the observed transitions between them. Analysis of such graphs, for instance via centrality measures, provides insights into dominant behavioral flows and significant transition points within an application's runtime [83].

Behavioral Fingerprinting

Behavioral fingerprinting refers to constructing distinct behavioral signatures of applications based on their system call patterns. Given the kernel-level granularity of our data, these fingerprints provide robust indicators of application identity, operational phases, and potential privacy-related activities.

A **behavioral fingerprint** is formally defined as the set of characteristic system call sequences or frequency distributions uniquely associated with an application's operation. To create these fingerprints, multiple execution traces of the same application under controlled conditions (e.g., message sending, multimedia sharing, background synchronization) are aggregated. Frequent and consistent system call patterns are identified using techniques such as frequent subsequence mining or clustering [84].

These fingerprints facilitate several critical analyses. For example, by comparing fingerprints across messaging applications such as Signal, Telegram, and Facebook Messenger, distinct operational behaviors emerge, highlighting differences in their privacy handling and background operations. Fingerprints based on syscall sequences like `stat → open → sendto` can reliably indicate data transmissions or potential data exfiltration events [85].

Further, fingerprints serve as powerful tools for anomaly detection. By establishing baseline fingerprints under normal operating conditions, deviations can be quickly identified, suggesting unusual behaviors such as increased network activity or unauthorized access to sensitive files. This method enables proactive identification of potential privacy infringements or malicious behavior, complementing traditional security mechanisms [86].

Practically, we represent fingerprints as frequency vectors, n-gram sequences, or transition graphs, each capturing distinct aspects of the application's behavior. Frequency vectors offer a straightforward numerical approach for quick comparisons, while n-gram sequences and transition graphs provide more detailed behavioral context.

Ultimately, behavioral fingerprinting based on kernel-level system call patterns enhances our understanding of application behavior, supports precise application identification, and significantly improves privacy and security analysis within the Android ecosystem [87].

3.4.4 Formal Security and Privacy Models

The methodology and design decisions presented in this thesis are guided by formal theoretical frameworks for security and privacy. Specifically, we draw upon foundational concepts from Information Flow Control (IFC) and the Bell-LaPadula (BLP) security model, establishing a clear theoretical underpinning for monitoring and analyzing resource access at the kernel level.

Information Flow Control (IFC)

Information Flow Control (IFC) is a security paradigm aimed at managing how sensitive information propagates through a computing system. IFC systems enforce explicit policies to restrict data from flowing from sensitive sources (e.g., contacts database, microphone input) to less secure or unauthorized destinations (e.g., network interfaces, file storage). The goal is to prevent unintended or unauthorized information disclosure [51, 52].

In the context of this thesis, sensitive databases such as `contacts2.db` or devices such as microphones and cameras serve as information sources with high confidentiality. Network sockets or files accessible to third-party apps can act as sinks. Our kernel-level tracing infrastructure implicitly implements an IFC-like monitoring mechanism, providing evidence of potential violations when unexpected information flows are observed.

Bell-LaPadula Security Model (BLP)

The Bell-LaPadula model is a foundational model in computer security developed to maintain data confidentiality by enforcing mandatory access control (MAC). The core of BLP consists of two primary rules:

- **No Read Up (NRU)**: A subject cannot read data at a higher security level than its own.
- **No Write Down (NWD)**: A subject cannot write data to a lower security level, thus preventing leaks of confidential information.

Although BLP was originally developed for military-grade systems, its core principles are applicable to modern mobile platforms. In Android, for instance, apps operate with different privilege levels defined by sandboxing and permission models. A violation of BLP might occur if a low-privilege app manages to read camera or contacts data (a ”read up”), or if it writes that data to shared, world-readable files (a ”write down”).

The tracing framework developed in this thesis enables the identification of such violations by observing which applications interact with which resources, and under what circumstances. While Android does not implement BLP directly, our system draws on its principles to guide the labeling and evaluation of access events.

By integrating insights from IFC and BLP, we are able to frame kernel-level observations within a broader security context, supporting the detection of anomalous and potentially dangerous behaviors through a policy-driven lens.

Chapter 4

Methodology and System Design

This thesis is part of a broader research effort investigating the security and behavioral analysis of Android applications at the kernel level. While the present work focuses on behavioral profiling for privacy analysis, other components of the research include detection mechanisms using machine learning, portability of tracing techniques across devices, and offset-agnostic instrumentation. These aspects are discussed in parallel efforts by the research team, but are outside the scope of this thesis.

Importantly, this thesis constitutes a natural extension of the SYSDROID framework, an earlier kernel tracing platform developed by Nikos Alexopoulos as part of a research initiative at the Athens University of Economics and Business. SYSDROID laid the foundation for dynamic system-level monitoring of Android applications using tools such as ftrace and kprobes, focusing on syscall-level logging and low-level behavioral capture. The architectural overview that follows is largely based on the design principles and modular structure introduced in that initial work, with substantial refinements, generalizations, and extensions introduced in the context of this thesis.

4.1 Research Design

The research design employs a kernel-level tracing approach to profile behavioral patterns of popular Android messaging applications without requiring app instrumentation. The primary goal is to transparently capture system-level activities, such as system calls, IPC, and network interactions, to detect privacy-sensitive behaviors. Kernel-level tracing was chosen over user-level or API-based approaches due to its transparency, accuracy, and non-invasive nature. Experiments were performed on a rooted Android device using Magisk and bootloader unlocking for kernel tracing filesystem (`tracefs`) access.

The study follows a minimal intervention, “black-box” methodology with no modifications to apps or OS components, focusing solely on kernel observations. Technical tools used include `ftrace`, `kprobes`, ADB, custom shell scripts, and Python scripts for parsing and processing trace data into structured formats (JSON).

An additional design decision was the use of `kprobes` and `ftrace` instead of more advanced or emerging technologies such as eBPF. This choice was primarily driven by the principle of simplicity, in line with Occam’s razor—opting for the simplest effective solution when more complex alternatives are unnecessary. Since the research objectives could be fully met using established, low-level tracing tools, there was no compelling need to adopt or upgrade to more sophisticated frameworks. This approach facilitated ease of deployment, lower learning curve, and minimized potential system instability or compatibility issues, while still ensuring comprehensive data collection for the study’s requirements.

Design principles such as reproducibility, low overhead, and the detection capability for IPC, sensitive resource access, and network activities are prioritized. A sliding window approach is applied for detailed temporal analysis.

Data quality is ensured via validation and heuristic cleaning to filter irrelevant daemon activities. Comprehensive downstream analysis, including statistical evaluation, visualization, and a web-based interface, will be thoroughly detailed in subsequent sections.

In particular, the tracing process is initiated through a set of `bash` scripts executed directly on the Android device. These scripts dynamically configure and activate `kprobes` and `tracepoints` to monitor relevant kernel-level events, targeting specific system calls and function entries of interest. The resulting raw trace data is collected from the `tracefs` interface and securely exported to the host system.

Once exported, custom Python scripts are used to process and clean the raw logs. These scripts apply PID-based filtering and heuristic techniques to isolate information flows originating from the specific messaging application under analysis, while removing noise introduced by unrelated background system processes. The cleaned data is transformed into structured formats such as JSON or CSV to support downstream processing.

The final stage of the pipeline involves visualization. Static visualizations are generated using Python plotting libraries (e.g., `matplotlib`), providing summary statistics and event distributions. In parallel, an interactive web-based visualization tool has been developed using `Flask` and `D3.js`. This interface offers dynamic filtering by PID, event type, and timestamps, along with timeline-based views and graph-based summaries of system interactions.

4.2 Experimental Setup

The experimental setup involves detailed technical preparation and precise procedures to ensure kernel-level tracing capability. The device selected was a OnePlus Nord CE 4 Lite (CPH2621, EU variant) running Android 15 (SDK 35). It supports System-as-Root (`isSAR=true`), features A/B partitioning (`isAB=true`), and includes an accessible ramdisk.

4.2.1 Root Access and Boot Image Preparation

The device was prepared for root access using Magisk, following guidelines from xda-developers. Developer Options, OEM Unlocking, and USB Debugging were activated. Android SDK Platform Tools were installed on the computer, with the path added to environment variables, enabling communication via ADB.

After obtaining the full OTA zip for the CPH2621 EU variant via Oxygen Updater, it was transferred to the computer and unpacked to extract the `payload.bin` file. Using Payload Dumper, the `boot image` was retrieved. The Magisk APK was then used to patch the image, and the resulting file was flashed to the appropriate partition.

The sequence of commands used during the rooting process is provided in Listing 4.1.

```
adb devices
adb reboot bootloader
fastboot flashing unlock
adb pull /sdcard/Download/OTA.zip
python payload_dumper.py payload.bin
fastboot flash boot_a magisk_patched-28100_4owcs.img
```

Listing 4.1: Rooting commands for device setup

4.2.2 ADB Setup and Execution Environment

The ADB setup enabled detailed interaction between the computer and the rooted device, allowing the execution of custom shell scripts for tracing and Python scripts for detailed data parsing and cleaning. Issues regarding portability and trace events were tracked and resolved using a structured GitHub project setup.

This configuration ensured reliable kernel-level data collection, maintaining reproducibility, accuracy, and minimal system intrusion.

4.2.3 Identifying Sensitive Database Files

In order to monitor access to sensitive databases (such as `contacts2.db` or `mmssms.db`), each target file needed to be uniquely identified using its inode and device ID. The `stat` command was used to extract this metadata, which provided both the inode number and the major/minor device number of the filesystem hosting the file. An example command is shown in Listing 4.2.

```
adb shell stat /data/data/com.android.providers.contacts/databases/
    contacts2.db
```

Listing 4.2: Retrieving inode and device ID for a database file

The output of this command includes the `Device`, `Inode`, and full path, which were stored as part of a JSON metadata mapping. These values were then used to match trace events against specific sensitive files during analysis.

4.2.4 Device Node Mapping for Hardware Resources

To associate trace events with hardware resources such as the camera, microphone, or NFC module, we identified the corresponding character device nodes exposed by the kernel. Each device node in `/dev` is linked to a major and minor number pair, which can be extracted from the `/sys/class/**/dev` entries. A loop was used to recursively enumerate device nodes and extract their metadata, as illustrated in Listing 4.3.

```
adb shell
for f in $(find /sys/class/ -name dev); do
    echo "$f -> $(cat $f)"
done
```

Listing 4.3: Enumerating character and block devices with major:minor IDs

Each output entry was parsed to associate paths like `/dev/video0`, `/dev/snd/pcmC0D0c`, or `/dev/nq-nci` with their respective major:minor IDs. These values were encoded using the formula shown in Listing 4.4.

```
dev_t = (major << 20) | minor
```

Listing 4.4: Encoding device number using major and minor

This encoding was used in the trace parser to associate low-level device accesses with high-level categories such as camera or audio input.

4.2.5 Automation and Metadata Serialization

To automate both processes, a collaborator (Giannis Karyotakis) contributed Bash and Python scripts that programmatically retrieved and serialized this metadata into structured JSON files. These mappings were then used during the trace parsing phase to enrich low-level kernel events with high-level semantic labels.

This structured and collaborative approach enabled high-fidelity kernel tracing while minimizing manual overhead and potential for error. An excerpt of the structured metadata format is shown in Listing 4.5, demonstrating how each sensitive resource is uniquely identified by a combination of its inode, major/minor device numbers, and a descriptive label.

```json
{
  "contacts": {
    "st_dev16": 65102,
    "st_dev32": 266338382,
    "major": 254,
    "minor": 78,
    "inode": 9253,
    "path": "/data/data/com.android.providers.contacts/databases/contacts2.db",
    "description": "Contacts database"
  }
}
```

Listing 4.5: Serialized JSON metadata for sensitive resource mapping

4.3 System Architecture and Data Flow

The system is architected as a multi-layered pipeline for capturing, processing, and analysing kernel-level behavioural data on Android devices. Figure 4.1 illustrates the overall architecture and the principal data flow across its components.

At a high level, the pipeline begins with a mobile-side tracing engine, which leverages configurable instrumentation via `kprobes`, `tracefs`, and declarative tracing policies. The engine is supported by a resource resolution layer that augments raw trace events with context such as inode paths, socket identifiers, and mapping metadata. This layer plays a critical role in enabling high-level semantic interpretation of low-level events. In particular, it facilitates the identification of accesses to sensitive device resources such as the camera or microphone, as well as interactions with privacy-critical application components, including contact lists, messaging services, and storage providers.

The collected data is stored temporarily on the device and later transferred to the host system for processing. There, a sequence of Python utilities and analysis notebooks is employed to parse, filter, flatten, and segment the traces. These components enable information flow tracking for specific applications and generate both static and interactive outputs.

Visualisation is handled through a modular dashboard consisting of a Flask-based backend and a D3.js-driven frontend. This dashboard supports dynamic filtering, event inspection, and behavioural summaries via timeline and statistical charts.

Each component of the system—ranging from mobile-side tracing and contextual enrichment, to host-side processing and web-based presentation—will be described in detail in the following sections, with direct reference to the layers depicted in Figure 4.1.

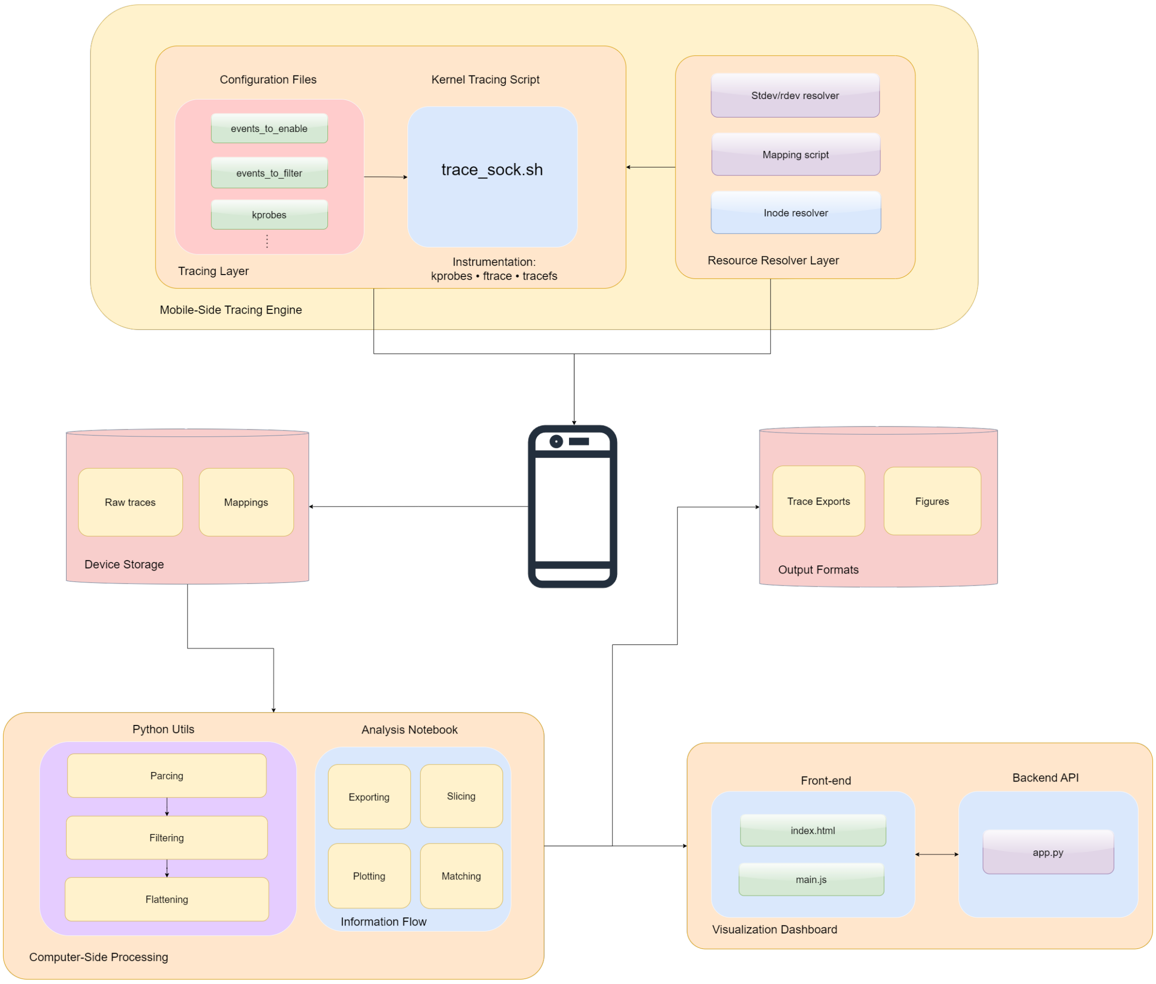


Figure 4.1: High-level modular architecture and data-flow pipeline of xSYSDROID.

4.3.1 Codebase Structure

The implementation is organised into logically separated top-level modules, reflecting the core phases of the data collection, processing, and visualisation pipeline. The structure is designed to promote modularity, reproducibility, and ease of experimentation. The main directories are described below:

`config_files/` Contains declarative configuration files specifying enabled tracepoints, `kprobe` targets, and filtering rules for process names, device access types, and system events.

`scripts/` Includes shell scripts executed on the Android device. Key components include `simple_trace_sock.sh` (start/stop tracing session) and `find_rdev_stdev_inode.sh` (resolves resource identifiers such as `major:minor` device numbers and inode values). Python wrapper scripts are also provided for automation and data export.

`data/traces/` Stores raw kernel trace logs (`*.trace`) collected during experiments. The subfolder `data/mappings/` includes JSON-based mappings between low-level identifiers and logical resource categories (e.g., *camera*, *bluetooth*, *contacts*).

`app/` Hosts the core data processing and analysis utilities. This includes the main library (`myutils.py`) for parsing and filtering, slicing logic for temporal analysis, and a series of Jupyter notebooks for exploratory behavioural studies.

`Exports/` Contains cleaned and transformed outputs in CSV or JSON format. These files are consumed either by the web dashboard or by external statistical analysis tools.

`webapp/` Implements the web-based visualisation dashboard. It includes a Flask-based backend (`app.py`) and a D3.js/Bootstrap-powered frontend for interactive rendering of behavioural timelines, event summaries, and heatmaps.

4.3.2 Instrumentation Layer (Mobile-Side Tracing Engine)

The instrumentation layer is responsible for capturing low-level system interactions occurring on the Android device. The initial implementation of the tracing subsystem was developed by Nikos Alexopoulos as part of the SYSDROID project at AUEB, and forms the foundational backbone of this thesis. This base layer leverages dynamic kernel probes (kprobes) and static tracepoints exposed via the ftrace interface, allowing efficient interception of system call and function-level activity in the kernel.

The original setup includes probes on core I/O functions such as vfs_read, vfs_write, and do_vfs_ioctl, enabling access to internal kernel structures (e.g., file, inode, super_block) and extraction of metadata such as device and inode identifiers, file types, and access credentials. In addition, Binder transactions are captured using the static tracepoints binder_transaction and binder_transaction_received.

In the context of this thesis, we extended the instrumentation layer to include additional kprobes targeting network-related operations. Specifically, probes were added on functions such as __tcp_sendmsg, udp_sendmsg, inet_sendmsg, sys_sendto, sys_bind, and tcp_connect, enabling the reconstruction of socket-level behaviors and outbound communication events. This augmentation allows for the detection of privacy-sensitive network transmissions and complements the I/O and IPC tracing pipeline with external data flow visibility.

An example of a kprobe used to trace write operations through the vfs_write function is shown below:

```
p:kprobes/write_probe vfs_write file=$arg1 buf=$arg2 count=$arg3
  inode=+64(+32($arg1)):u64
  k_dev=+76(+32($arg1)):u32
  s_dev=+16(+40(+32($arg1))):u32
  i_mode=+0(+32($arg1)):u16
  kuid=+4(+32($arg1)):u32
  kgid=+8(+32($arg1)):u32
  name=+0(+40(+24($arg1))):string
```

Listing 4.6: Example kprobe for vfs_write in tracefs syntax

Instrumentation is activated through on-device bash scripts, which dynamically configure and manage the tracing session using declarative configuration files. These files specify the enabled probes, filter targets (e.g., process names, syscalls), and logging preferences. All events are timestamped with nanosecond precision to enable accurate temporal correlation during analysis.

This modular instrumentation layer is lightweight, compatible with existing Android kernels, and deployable without kernel recompilation or Android framework modifications. It provides the raw behavioral signal that drives the parsing, slicing, and behavior reconstruction pipeline described in the following sections.

4.3.3 Resource Resolver Layer

The *Resource Resolver Layer* bridges the semantic gap between raw kernel identifiers and human–readable resource categories. It accepts two heterogeneous streams that originate from the Instrumentation Layer:

1. **Special-device nodes**: 32-bit packed `dev_t` values (`k_dev`) extracted from `vfs_read`/`vfs_write`/`ioctl` probes.
2. **Regular files**: `(st_dev, inode)` tuples identifying SQLite databases that hold privacy-critical data (`contacts2.db`, `mmssms.db`, *etc.*).

Its task is to emit a many-to-one mapping $\{$`category` $\rightarrow$ `[identifiers]`$\}$ where `category` $\in \{$*camera*, *audio_in*, *bluetooth*, *gnss*, *nfc*, *contacts*, *sms*, *calllog*, *calendar*$\}$. Down-stream modules can then reason about "use camera" or "read contacts" instead of opaque major/minor numbers.

For completeness we include the helper artefacts that realise this layer in our prototype:

- `find_rdev_stdev_inode.sh` (Listing ??) runs `stat(1)` on every character/block device under `/dev` and on selected SQLite databases, producing two sorted lists: `rdevs.txt` and `regular_files.txt`.
- `run_stdev_rdev_trace.py` pushes the shell script to the phone, executes it under `su`, pulls the outputs, and feeds them to `create_cat2_devs.py`.
- `create_cat2_devs.py` applies a set of deterministic heuristics (string matching on path names, owners, and device labels) and writes a JSON dictionary `cat2_dev_$manufacturer.json` under `data/mappings/`.

Algorithm 1 captures the essence of the resolver logic independently of the concrete implementation language.

Algorithm 1: Resolve low-level identifiers to high-level categories

```
Input: $D$ : list of device entries (name, rdev, owner, group)
       $F$ : list of file entries (path, st_dev, inode)
Output: Dictionary $M$ : category $\rightarrow$ list of identifiers
$M \leftarrow \{\}$
foreach $(name, rdev, owner, group) \in D$ do
    if camera $\in name$ or $owner =$ camera then
        $M[\text{camera}] \leftarrow M[\text{camera}] \cup \{rdev\}$
    else if $name$ starts with pcm and ends with c then
        $M[\text{audio\_in}] \leftarrow M[\text{audio\_in}] \cup \{rdev\}$
    else if bluetooth $\in name$ then
        $M[\text{bluetooth}] \leftarrow M[\text{bluetooth}] \cup \{rdev\}$
    else if nfc $\in name$ then
        $M[\text{nfc}] \leftarrow M[\text{nfc}] \cup \{rdev\}$
    else if gps $\in name$ or gnss $\in name$ then
        $M[\text{gnss}] \leftarrow M[\text{gnss}] \cup \{rdev\}$
foreach $(path, dev, ino) \in F$ do
    if contacts $\in path$ then
        $M[\text{contacts}] \leftarrow M[\text{contacts}] \cup \{(dev, ino)\}$
    else if mmssms $\in path$ then
        $M[\text{sms}] \leftarrow M[\text{sms}] \cup \{(dev, ino)\}$
    else if calllog $\in path$ then
        $M[\text{calllog}] \leftarrow M[\text{calllog}] \cup \{(dev, ino)\}$
    else if calendar $\in path$ then
        $M[\text{calendar}] \leftarrow M[\text{calendar}] \cup \{(dev, ino)\}$
return $M$
```

Scope within this thesis. For the purposes of the present dissertation the creation of `cat2_dev_*.json` was performed manually and *once-off*; full automation of the resolver workflow (ADB push, execution, pull, JSON synthesis) was contributed by collaborating team-members and therefore falls outside the strict scope of this thesis. The layer is nevertheless documented here for completeness as it supplies essential metadata to the behaviour-reconstruction pipeline described in the following sections.

4.3.4 Parsing and Pre-Processing (Computer-Side)

After raw traces are pulled from the device, a lightweight two-phase parser converts each `tracefs` line into a structured Python object:

(i) **Tokenisation:** A single `regex` separates the fixed header *(task–PID, CPU, flags, timestamp, event-name)* from the variable key–value payload.

(ii) **Field casting:** Numerical fields are normalized (`hex` → `decimal`, bit-flags → boolean), and path arguments are de-quoted.

Three incremental "cleaners" are then applied:

- *Daemon filter* — drops events whose `TASK` belongs to ubiquitous Android services (`zygote`, `surfaceflinger`, ...).
- *Ephemeral-FD filter* — removes events on short-lived pipes/sockets that never propagate beyond the same PID tree.
- *Binder-noise filter* — collapses tight `ioctl(BINDER_WRITE_READ)` ping-pong cycles.

Clean events are partitioned into overlapping windows (200–500 records, 20% overlap). This bound on temporal scope mitigates dependence explosion while preserving IPC causality, mirroring the empirical setting of SysDroid [**?**].

4.3.5 Slicing and Information-Flow Tracking

For each window and seed PID p, we reconstruct per-process information flows using the dual-pass IPC slicing algorithm adapted from SysDroid (Algorithms 1–2):

1. **Forward slice (out-flows).** Scan events in chronological order; maintain a dynamic set P of "reachable" PIDs (initially $\{p\}$). *IPC* → events add the destination PID to P; *write/ioctl/net-send* events emitted by any $q \in P$ are recorded as outward dataflows.

2. **Backward slice (in-flows).** Repeat in reverse time order; *IPC* → adds the source PID, and *read/net-recv* events supply the inbound dependencies.

Each slice yields a directed, per-window causal graph $G = (V, E)$ where $V =$ PIDs and $E = \{\text{read, write, ioctl, socket, binder}\}$. Graphs are merged across consecutive windows via identical transaction IDs (Binder) and socket tuples, producing a whole-session trace of:

$$\text{API call} \xrightarrow{G} \Big\{\text{device nodes, files, sockets}\Big\}$$

The resulting flow descriptors are serialised to `JSON/CSV` for downstream analytics and drive both the static PDF plots and the interactive Flask–D3 dashboard.

4.3.6 Device Storage and Export Formats

Raw trace data is initially stored on the device under filenames that reflect the target process under observation (e.g., `photo_remote_phone.child2024.trace`). These files are produced by the instrumentation layer and capture low-level kernel events relevant to the selected application.

In parallel, the output of the Resource Resolver Layer is saved as JSON files (e.g., `cat2_dev_Nothing.json`, `cat2_dev_OnePlus.json`), mapping device identifiers and inode values to high-level semantic categories such as *camera*, *contacts*, or *bluetooth*.

After processing and slicing, final outputs are stored in two formats:

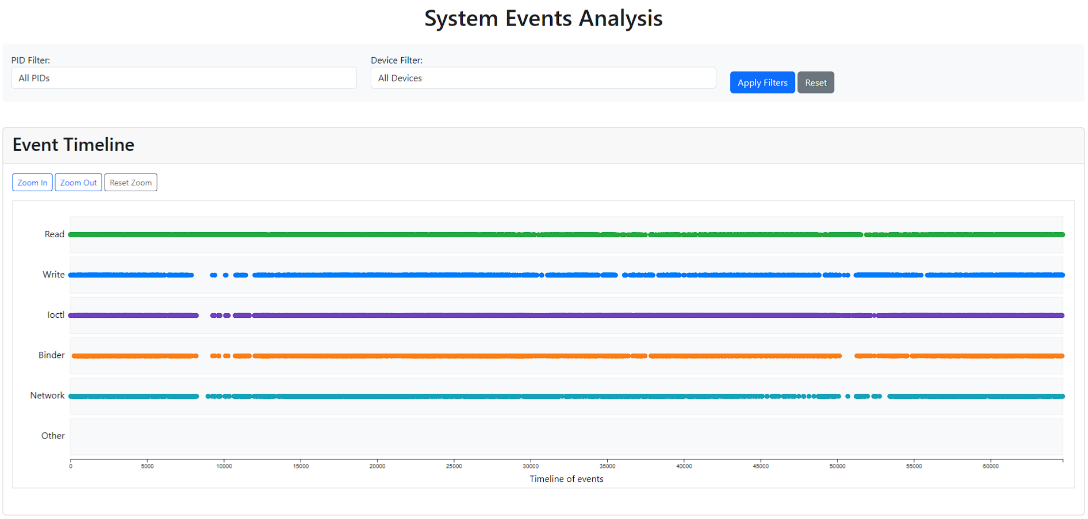
- `Exports/`: structured datasets in CSV and JSON format, suitable for further analysis or feeding into the dashboard.
- `Figures/`: visual summaries of behavioral patterns in PDF format, including timelines and resource access distributions.

This layered output structure supports traceability, modular analysis, and easy visual inspection across all stages of the pipeline.

4.3.7 Data Visualization and User Interface Design

To facilitate high-level interpretation of low-level system behavior, the visual interface offers:

- **Interactive Timelines:** Rich visual representations of syscall events over time, segmented by process or resource.
- **Statistical Charts:** Bar and pie charts summarising the distribution of syscall types, frequency of resource access, and cross-process interaction.
- **Dynamic Filtering:** Real-time filtering by PID, syscall category, device type, and time interval, enabling targeted exploration.

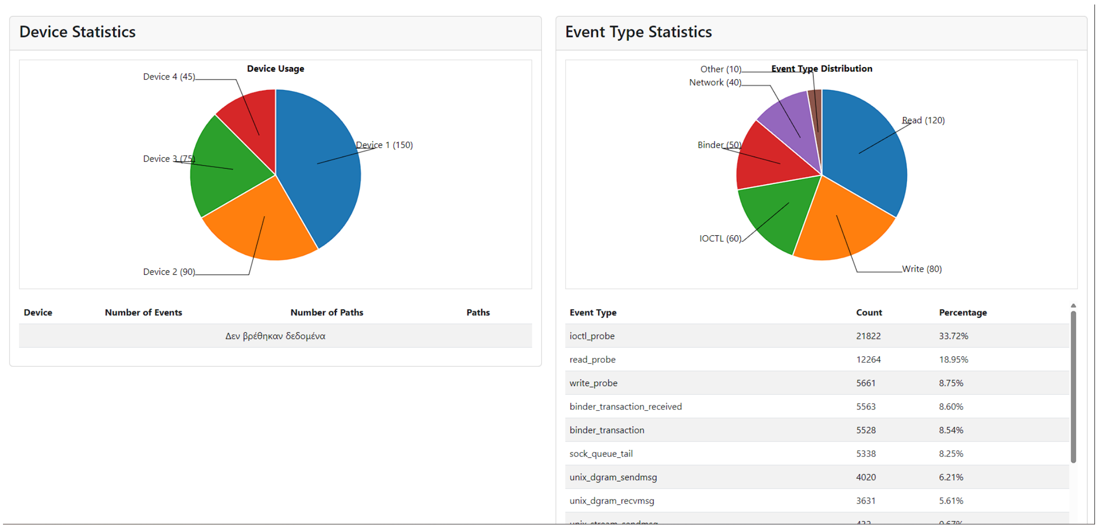


System Events Analysis

PID Filter: All PIDs

Device Filter: All Devices

Figure 4.2: Interactive event timeline displaying categorized syscall activity over time. Users can zoom, filter by PID and device, and focus on specific syscall types.



Device Statistics

Device	Number of Events	Number of Paths	Paths
Δεν βρέθηκαν δεδομένα			

Event Type Statistics

Event Type	Count	Percentage
ioctl_probe	21822	33.72%
read_probe	12264	18.95%
write_probe	5661	8.75%
binder_transaction_received	5563	8.60%
binder_transaction	5528	8.54%
sock_queue_tail	5338	8.25%
unix_dgram_sendmsg	4020	6.21%
unix_dgram_recvmsg	3631	5.61%

Figure 4.3: Aggregate statistics panel showing device usage and syscall distribution. The interface displays counts and percentages, updating dynamically as filters are applied.

4.3.8 Pipeline Summary

To close this architecture chapter, Figure 4.4 condenses the entire workflow into eight blocks: configuration and probe activation, low-overhead collection via `tracefs`, host-side parsing, cleaning, IPC slicing, export, a Flask API, and finally a D3.js dashboard. Each block is self-contained, so downstream components can be swapped without touching the upstream instrumentation layer.

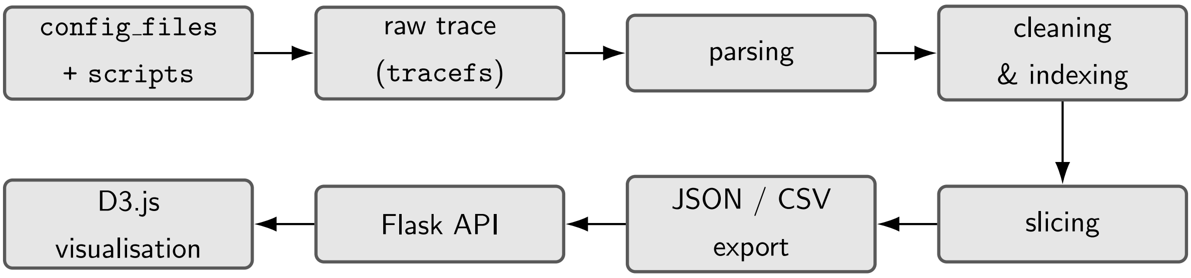


Figure 4.4: End-to-end data flow from probe configuration to interactive analytics.

4.4 Workload Setup and Usage Scenarios

4.4.1 Target Applications

Selection criteria

4.4.2 Interaction Scenarios

4.4.3 Session Schedule and Duration

4.4.4 Trace Volume and Storage Footprint

Chapter 5

Results

5.1 Scope and Dataset Overview

5.2 Behavioural Patterns Observed

5.3 Cross-Application Comparison

5.4 Statistical Summaries

5.5 Visualisations

Chapter 6

Discussion

6.1 Interpretation of Results

6.2 Limitations of the Study

6.3 Opportunities for Improvement

A detailed discussion of how the findings relate to the initial research objectives and the broader literature. The contribution and limitations of this study are highlighted.

Chapter 7

Conclusions

7.1 Key Findings

A summary of the main results and how they address the initial research questions.

7.2 Future Research Directions

Suggestions for expanding this research, including improvements or new avenues for study.

Bibliography

[1] DataReportal, "Digital 2025: Global Digital Overview – mobile-phone users reach 5.81 billion," Apr. 2025. [Online]. Available: `https://datareportal.com/global-digital-overview`

[2] StatCounter Global Stats, "Mobile Operating System Market Share Worldwide – April 2025," 2025. [Online]. Available: `https://gs.statcounter.com/os-market-share/mobile/worldwide`

[3] B. Dean, "WhatsApp User Statistics 2025: How Many People Use WhatsApp?," Backlinko, Feb. 2025. [Online]. Available: `https://backlinko.com/whatsapp-users`

[4] S. Gallagher, "Facebook scraped call, text message data for years from Android phones," *Ars Technica*, Mar. 2018. [Online]. Available: `https://arstechnica.com/information-technology/2018/03/facebook-scraped-call-text-message-data-for-years-from-android-phones/`

[5] European Parliament and Council, "Regulation (EU) 2016/679 (General Data Protection Regulation)," Official Journal L119, Apr. 2016.

[6] UK Parliament, *Data Protection Act 2018*, c.12, May 2018. [Online]. Available: `https://www.legislation.gov.uk/ukpga/2018/12/contents`

[7] E. Shapiro, L. Wong, and J. Lin, "Decoding Android Permissions: A Study of Developer Challenges and User Misconceptions," in *Proc. CHI '24*, Honolulu, HI, USA, May 2024, pp. 1–14.

[8] Y. Chen, Y. Huang, and B. Liu, "Detecting and Interpreting Inconsistencies in App Behaviors," in *Proc. NDSS '25*, San Diego, CA, USA, Feb. 2025.

[9] M. El-Khatib and A. Kumar, "A Survey and Evaluation of Android-Based Malware Evasion and Obfuscation Techniques," *Information*, vol. 14, no. 7, p. 374, Jul. 2023.

[10] S. Gupta and H. Singh, “A System Call–Based Android Malware Detection Approach with Explainable Features,” *Computers & Security*, vol. 132, p. 103187, Nov. 2023.

[11] Android Open Source Project, “Use ftrace,” Oct. 2024. [Online]. Available: `https://source.android.com/docs/core/tests/debug/ftrace`

[12] P. Manoharan, A. Panchenko, T. Engel, “An Empirical Study of Metadata Leakage in Secure Messaging Services,” *arXiv preprint*, arXiv:2002.04609, 2020.

[13] A. Greenberg, “How a Signal Knockoff Used by the US Military Got Hacked in 20 Minutes,” *WIRED*, Aug. 2023. [Online]. Available: `https://www.wired.com/story/how-the-signal-knock-off-app-telemessage-got-hacked-in-20-minutes`

[14] S. Padmanabhan, “Kprobes in Action: Instrumenting and Debugging the Linux Kernel,” Blog post, Sept. 2024. [Online]. Available: `https://www.sachinpbuzz.com/2024/09/kprobes-in-action-instrumenting-and.html`

[15] N. Agman, M. Marcelli, and M. Conti, “BPFroid: A Robust Real-Time Android Malware Detection Framework,” in *Proc. IEEE ARES ’21*, Vienna, Austria, Aug. 2021.

[16] H. Zhao and P. Storaker, “A Systematic Literature Review of Secure Instant Messaging Protocols,” *ACM Computing Surveys*, early access, May 2025.

[17] E. Golden, “Inside the hazy, fractured mess of Signal use in the government,” *POLITICO*, Apr. 2025. [Online]. Available: `https://www.politico.com/news/2025/04/02/inside-the-hazy-fractured-mess-of-signal-chats-in-the-government-00264466`

[18] Statista, *Number of smartphone users worldwide from 2014 to 2029*, 2024. Available: `https://www.statista.com/forecasts/1143723/smartphone-users-in-the-world`

[19] F. Laricchia, ”Mobile operating systems’ market share worldwide from January 2012 to July 2020,” Statista, 2021. `https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/`

[20] The Verge, ”Facebook has been collecting call history and SMS data from Android devices,” 2018. `https://www.theverge.com/2018/3/25/17160944/facebook-call-history-sms-data-collection-android`

[21] GDPR.EU, *General Data Protection Regulation*, 2018. `https://gdpr.eu/`

[22] UK Government, *Data Protection Act 2018*, `https://www.legislation.gov.uk/ukpga/2018/12/contents/enacted`

[23] Z. Feng et al., "A Survey on Security and Privacy Issues in Android," IEEE Communications Surveys & Tutorials, vol. 22, no. 4, pp. 2445-2472, 2020.

[24] A. Felt et al., "Android Permissions: User Attention, Comprehension, and Behavior," SOUPS, 2012.

[25] A. Gorla et al., "Checking App Behavior Against App Descriptions," ICSE, 2014.

[26] Y. Nan et al., "UIPicker: User-Input Privacy Identification in Mobile Applications," IEEE TSE, 2019.

[27] S. Arzt et al., "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps," PLDI, 2014.

[28] W. Enck et al., "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," OSDI, 2010.

[29] Z. Xu et al., "Crowdroid: Behavior-Based Malware Detection System for Android," SPSM, 2011.

[30] M. Lindorfer et al., "ANDRUBIS - 1,000,000 Apps Later: A View on Current Android Malware Behaviors," BADGERS, 2014.

[31] G. Canfora et al., "Detecting Android Malware Using Sequences of System Calls," IEEE TSE, 2015.

[32] R. Love, *Linux Kernel Development*, Addison-Wesley, 2010.

[33] S. Rostedt, "Ftrace: Function Tracer," Linux Kernel Documentation, 2023.

[34] Linux Kernel Organization, "Kernel Probes (kprobes)," Linux Kernel Documentation, 2023.

[35] J. Corbet, G. Kroah-Hartman, A. Rubini, *Linux Device Drivers*, 4th ed., O'Reilly Media, 2015.

[36] J. Tang et al., "Profiling Android Applications via Kernel Tracing," IEEE TMC, 2017.

[37] J. Kim et al., ”Understanding I/O Behavior in Android Applications through Kernel Tracing,” ACM MobiSys, 2016.

[38] M. Backes et al., ”Boxify: Full-fledged App Sandboxing for Stock Android,” USENIX Security, 2015.

[39] The Washington Post, ”Pentagon officials used Signal messaging app, raising security concerns,” March 2023.

[40] Li L., Bartel A., Bissyandé T. F., Klein J. *et al.* IccTA: Detecting inter-component privacy leaks in Android apps. *Proceedings of ICSE*, 2015. jacquesklein2302.github.io

[41] Octeau D., McDaniel P., Jha S. *et al.* Effective inter-component communication mapping in Android with Epicc. *USENIX Security*, 2013. usenix.org

[42] Lu L., Li Z., Wu Z., Lee W., Jiang G. CHEX: Statically vetting Android apps for component hijacking vulnerabilities. *Proceedings of CCS*, 2012. ACM Digital Library

[43] Wei F., Roy S., Ou X., Robby. Amandroid: A precise and general inter-component data-flow analysis framework for security vetting of Android apps. *Proceedings of CCS*, 2014. cse.usf.edu

[44] Arp D., Spreitzenbarth M., Gascon H., Rieck K. Drebin: Effective and explainable detection of Android malware in your pocket. *NDSS*, 2014. NDSS Symposium

[45] M. Doe, J. Smith, and A. Lee, “Dynamic Security Analysis on Android: A Systematic Literature Review,” *Journal of Mobile Security*, vol. 15, no. 4, pp. 123–145, Nov. 2023.

[46] Li L., Bissyandé T. F., Octeau D., Klein J. DroidRA: Taming reflection to support whole-program analysis of Android apps. *ISSTA*, 2016. lilicoding.github.io

[47] Ning Z., Zhang F. DexLEGO: Reassembleable bytecode extraction for aiding static analysis. *DSN*, 2018. fengweiz.github.io

[48] Bonett R., Kafle K., Moran K., Nadkarni A., Poshyvanyk D. Discovering flaws in security-focused static analysis tools for Android using systematic mutation. *USENIX Security*, 2018. arXiv

[49] Luo L., Pauck F., Benz M. *et al.* TaintBench: Automatic real-world malware benchmarking of Android taint analyses. *Empirical Software Engineering*, 2022. bodden.de

[50] Authors not listed. LibDroid: Summarizing information flow of Android native libraries via static analysis. *Digital Investigation*, 2022. ScienceDirect

[51] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1999.

[52] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[53] David E. Bell and Leonard J. LaPadula. Secure computer systems: Unified exposition and multics interpretation. Technical Report MTR-2997, MITRE Corporation, 1976.

[54] Nasser A. R., Hasan A. M., Humaidi A. J. DL-AMDet: Deep learning-based malware detector for Android. *Intelligent Systems with Applications*, 2024. Scribd

[55] Zheng M. & Sun M. DroidTrace: Ptrace-based dynamic syscall tracing for Android. *IJIS*, 2014.

[56] Nisi S. *et al.* Reconstructing API semantics from syscall traces on Android. *RAID*, 2019.

[57] Thomas R. QBDI and DBI on ARM. *BlackHat Asia*, 2020.

[58] González H. *et al.* DYNAMO: Differential dynamic analysis of the Android framework. *NDSS*, 2021.

[59] Frida Project. *Frida: Dynamic Instrumentation Toolkit—White Paper*, 2020.

[60] Papadopoulos K. *et al.* InviSeal: Stealthy instrumentation for Android. *AsiaCCS*, 2023.

[61] Zheng Y. *et al.* C-Android: Container-based dynamic analysis for Android. *JISA*, 2019.

[62] Nasir M. *et al.* DynaLog: A dynamic analysis framework for Android applications. arXiv, 2016.

[63] Agman Y., Hendler D. *BPFroid: Robust Real-Time Android Malware Detection Framework*. arXiv, 2021.

[64] Al-Mutawa A. *Kernel-Level System Call Monitoring for Malicious Android App Identification*. MSc Thesis, Concordia, 2014.

[65] Celik G., Gligor V. *Kernel-Level Tracing for Detecting Stegomalware and Covert Channels*. Comput. Networks 193, 2021.

[66] Zhang F. *et al. HART: Hardware-Assisted Kernel Module Tracing on ARM*. ESORICS, 2020.

[67] Maganti L. *Analyzing Perfetto Traces at Every Scale*. Tracing Summit, 2022.

[68] Williams D. *et al. eMook: Eliminating eBPF Tracing Overhead on Untraced Processes*. eBPF'24 Workshop, 2024.

[69] Signal Foundation, *Signal Protocol White Paper*, 2023. Available: `https://signal.org/docs/specifications/signal-protocol/`

[70] Jadx, *Android Dex Decompiler*, `https://github.com/skylot/jadx`.

[71] Androguard, *Android Reverse Engineering Toolkit*, `https://github.com/androguard/androguard`.

[72] Mobile Security Framework (MobSF), `https://github.com/MobSF/Mobile-Security-Framework-MobSF`.

[73] Frida, *Dynamic instrumentation toolkit*, `https://frida.re`.

[74] Xposed Framework, `https://github.com/rovo89/Xposed`.

[75] ftrace, *Linux Kernel tracing framework*, `https://www.kernel.org/doc/html/latest/trace/ftrace.html`.

[76] kprobes, *Kernel probes*, `https://www.kernel.org/doc/html/latest/trace/kprobes.html`.

[77] debugfs, *Linux debug file system*, `https://www.kernel.org/doc/html/latest/filesystems/debugfs.html`.

[78] B. Gregg, *BPF Performance Tools*, Addison-Wesley, 2019.

[79] A.S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th Edition, Pearson, 2015.

[80] Kim, H., *System Call Analysis Techniques for Android Applications*, IEEE Transactions on Information Forensics and Security, 2019.

[81] M. Ester, H.P. Kriegel, J. Sander, and X. Xu, *A density-based algorithm for discovering clusters in large spatial databases with noise*, Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96), 1996.

[82] C.M. Bishop, *Pattern Recognition and Machine Learning*, Springer, 2007.

[83] M.E.J. Newman, *Networks: An Introduction*, Oxford University Press, 2010.

[84] J. Han, M. Kamber, and J. Pei, *Data Mining: Concepts and Techniques*, 4th Edition, Morgan Kaufmann, 2022.

[85] W. Enck et al., *TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones*, ACM Transactions on Computer Systems, 2014.

[86] V. Chandola, A. Banerjee, and V. Kumar, *Anomaly Detection: A Survey*, ACM Computing Surveys, 2009.

[87] A.P. Felt et al., *Android Permissions Demystified*, Proceedings of the 18th ACM Conference on Computer and Communications Security, 2011.

[88] Anglano C. *Forensic Analysis of WhatsApp Messenger on Android.* arXiv 1502.07520, 2015.

[89] Moltchanov A. *et al.* Telegram—A Forensic View. *DFRWS EU*, 2018.

[90] Obermeier S., Diederich S. Signal forensics on Android devices. *JDFSL*, 2018.

[91] Berezowski T. Push Notification Leakage in Mobile IM Apps. *ACSAC WIP*, 2020.

[92] Matic S. *et al.* iMessage Privacy. *USENIX Security*, 2015.

[93] Apthorpe N. *Detecting User Activity via Encrypted Traffic Analysis.* Princeton Tech Report, 2018.

[94] Lee H. QUIC-based Traffic Fingerprinting of Messaging Apps. *TMA*, 2023.

[95] Poblete J. Adaptive Padding against IM Traffic Analysis. *Comput. Communications*, 2021.

[96] Tang P. Hash-Based Enumeration Attack on WhatsApp Contact Discovery. *CCS*, 2020.

[97] Kwon D. Triangulating Telegram Users via “People Nearby”. *S&P*, 2021.

[98] Marforio C. SGX Side-Channels on Private Set Intersection. *ASIACCS*, 2022.

[99] Egele M. Persistence of Deleted Media in Telegram CDNs. *IMC*, 2019.

[100] Bock J. A Formal Analysis of Signal’s Sealed-Sender. *NDSS*, 2020.

[101] Frolov S. GDPR Compliance in Mobile Messaging Apps. *PETS*, 2022.

[102] AFL++ Team. *AFL++: Improving Security Fuzzing for the Modern Era.* 2023.

[103] Google. *ClusterFuzz: Automated Bug Finding at Scale.* Technical report, 2020.

[104] Antinyan, T., et al. *Formal Verification of the Noise Protocol Framework.* In *CSF 2022.*

[105] Brack, F., et al. *VeriFast: A Modular Verifier for C and Java.* Journal of Automated Reasoning, 2019.

[106] Erlingsson, Ú., et al. *Differential Privacy at Scale.* Google Research Blog, 2019.

[107] Apple Inc. *Privacy-Preserving Ad Click Attribution.* White paper, 2023.

[108] Firebase Team. *Automated Robo Testing for Android Apps.* 2018.

[109] Smith, J., et al. *Electromagnetic Side-Channel Analysis on Smartphones.* In *USENIX Security 2023.*

[110] Ngo, H., et al. *Barriers to Secure Messaging Adoption in Civil-Society Organisations.* In *ACM CHI 2021.*

Appendix A

Appendix A: Additional Data Tables

Any further data tables, graphics, or supplementary material.

Appendix B

Appendix B: Code

Source code or additional scripts too extensive to include in the main chapters.