

**ΟΙΚΟΝΟΜΙΚΟ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΑΘΗΝΩΝ**



**ATHENS UNIVERSITY
OF ECONOMICS
AND BUSINESS**

Athens University of Economics and Business

Department of Management Science and Technology

Undergraduate Thesis

Hybrid Behavioral Profiling of Popular Messaging Apps via Custom Kernel-Level Tracing

Student:

Foivos - Timotheos Proestakis

Student ID: 8210126

Supervisors:

Prof. Diomidis Spinellis

Dr. Nikolaos Alexopoulos

Submission Date:

June 16, 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 [?]. Among the dominant mobile platforms, Android—an open-source operating system developed by Google—holds a stable global market share of approximately 75 % [?]. 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 [?].

Incidents such as Facebook’s unauthorized collection of SMS texts and call logs from Android devices [?] 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 [?, ?]. Despite these efforts, Android’s permission model often fails in practice—users frequently misinterpret the scope of the privileges they grant, inadvertently exposing sensitive data [?].

End-to-end encryption (E2EE) provides strong guarantees for content confidentiality, yet its effectiveness ultimately depends on correct implementation, device-level protections, and careful metadata handling [?, ?]. Messaging apps may still leak information through patterns such as message timing or background system activity—even when message payloads are encrypted.

Capturing and understanding this low-level behavior requires runtime analysis. Kernel-level tracing is a powerful technique for that purpose: tools such as `ftrace` and `kprobes` record system-call activity with minimal overhead, offering fine-grained visibility into how an application interacts with operating-system resources [?, ?]. Yet only a handful of studies have focused on applying such tracing specifically to messaging apps [?], despite their widespread and sensitive use—including by government and military officials [?].

This thesis investigates three core aspects: (i) whether declared permissions align with runtime resource usage, (ii) the extent of background access to privacy-sensitive data, and (iii) behavioural differences between privacy-centric and commercial messaging apps. The precise research questions are listed in Section 1.3.

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 [?]. 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:** Presents the rise of mobile messaging and privacy risks, highlights limits of Android permissions and E2EE, motivates kernel-level tracing, and states the thesis goals, research questions, and scope.
- **Chapter 2 – Related Work:** Reviews static analysis, user-space and kernel-space dynamic instrumentation, and prior privacy studies on messaging apps, pinpointing the gap in systematic runtime profiling of encrypted services.
- **Chapter 3 – Technical Background:** Summarises Android/Linux architecture, Binder IPC, SELinux, and the system-call interface; explains `ftrace/kprobes` and the semantic gap between kernel events and user actions.
- **Chapter 4 – Methodology & System Design:** Describes device setup, tracing configuration, data-processing pipeline, feature extraction, and the web-based visualisation dashboard.

- **Chapter 5 – Results:** Reports empirical findings for WhatsApp, Telegram, and Signal: syscall frequencies, permission-behaviour mismatches, and comparative visuals of privacy-relevant patterns.
- **Chapter 6 – Discussion:** Relates results to the research questions, examines implications for transparency and metadata leakage, and outlines methodological limitations and improvements.
- **Chapter 7 – Conclusions:** Recaps contributions, emphasises the value of kernel tracing for privacy audits, and proposes future work (more apps, automated behaviour classification, multi-layer tracing).
- **Appendix A – Additional Data Tables:** Supplementary tables and charts supporting Chapter 5.
- **Appendix B – Code:** Selected scripts and configuration files used for tracing and analysis.

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 [?]. 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 *Droid-Bench* suite it reaches 93 % recall and 86 % precision—outperforming both academic and commercial tools [?]¹—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** [?] and **CHEX** [?]*—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 [?].

(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 % [?]. **DexLEGO** reassembles byte-code collected at run time [?], while **LibDroid** summarises taint propagation inside native libraries [?]; 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 [?]. Deep networks now dominate: **DL-AMDet** combines CNN-BiLSTM embeddings with an auto-encoder to reach 99.9 % accuracy on public datasets [?]. 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 [?]. 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 [?]. 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 [?].

(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 [?]. **DYNAMO** layers systematic diff-based analysis on top of Frida to study framework evolution across Android versions [?], while **InviSeal** hardens Frida against detection, shrinking overhead to $< 3\%$ on UI benchmarks [?]. 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 [?]. Emulation-centred pipelines (e.g. **DynaLog** or DroidBox derivatives) automate large-scale malware triage by collecting ~ 100 runtime features in parallel VMs [?]. 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

2.3.1 Tracing Infrastructures

Kernel hooks sit beneath both the Java/ART runtime and Linux user-space, capturing syscalls, Binder traffic, scheduler switches and network I/O with negligible perturbation. Their vantage point is indispensable once payloads are encrypted in user space—TLS bursts, Binder IPC patterns and power wakes are often the only artefacts left to observe. Prior art clusters into four tool-centric strands:

- (a) **eBPF- and *ftrace*-based collectors.** *BPFroid* streams per-UID syscalls, Binder calls and network events from stock kernels to a malware detector at ~3 Earlier, *ID-Syscall* logged syscall sequences via a loadable module on Android 4.x, showing that rich behavioural signals are already available on commodity devices—albeit exploited only for coarse binary classification.
- (b) **Network-centric and covert-channel tracing.** Celik & Gligor detect stegomalware solely from timing and packet-size anomalies in vanilla *ftrace* logs, proving that scheduler, socket and power events suffice to expose covert channels—yet Binder traffic is omitted.
- (c) **Hardware-assisted monitors.** *HART* streams ARM ETM traces of proprietary kernel modules almost for free, but ETM is often fused-off in production phones and lacks user-process context.
- (d) **Production-scale, low-overhead tracing.** Google’s *Perfetto* unifies *ftrace*, Binder and user-space counters into multi-GB, SQL-queryable traces for start-up and jank analysis, while *eMook* adds per-PID eBPF filters, cutting system cost by 85

2.3.2 Behaviour Reconstruction & Transparency

After tracing primitives matured, attention shifted to re-assembling high-level behaviours from low-level footprints:

- (e) **Cross-layer behaviour reconstruction.** *CopperDroid* fuses syscalls with Binder payloads via Virtual-Machine Introspection; *Dagger* builds syscall–IPC provenance graphs; and *SysDroid* slices kernel traces along Binder causality,

covering privileged system apps on real devices with minimal overhead. Multi-layer instrumentation can reach finer granularity but at $5\text{--}40 \times$ runtime overhead, making it lab-only :contentReference[oaicite:0]index=0.

- (f) **Dynamic taint and information-flow trackers.** TaintDroid, TaintART, ARTist and FSAFlow instrument the ART/Java runtime to propagate taints at object level; they are precise but incur noticeable slowdown, require framework changes and can be detected by malware—limitations avoided by pure kernel tracing :contentReference[oaicite:1]index=1.
- (g) **Provenance-system perspectives.** Enterprise frameworks such as CamFlow, Hi-Fi and Spade record lightweight syscall provenance for APT detection on desktop systems; Android’s semantic gap hampers their direct reuse, but Binder-aware kernel tracing (e.g., SysDroid) can supply the missing context and extend host-based detection to mobile :contentReference[oaicite:2]index=2.
- (h) **Built-in transparency mechanisms.** Since Android 12, status-bar indicators flag camera/mic use by third-party apps, yet they exclude system services and can be bypassed via permission abuse, overlays or framework bugs. Kernel-level device-file tracing, as done by SysDroid, closes these blind spots :contentReference[oaicite:3]index=3.

Synthesis. Kernel tracing offers a hard-to-tamper, always-on lens but must still tame five hurdles: *semantic gap*, *opaque data*, *privilege requirements*, *data volume*, *detectability*. Recent work narrows these by (i) pairing syscalls with Binder provenance, (ii) granting BPF rights via SELinux instead of root, (iii) summarising high-rate events in-kernel, and (iv) obscuring probes via ETM/PTM or uprobe randomisation—pushing low-overhead, whole-system reconstruction onto stock phones.

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

[?]. 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 [?, ?]. While modern versions encrypt local backups, notification side-channels (e.g. Firebase tokens) remain an open avenue [?].

(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 [?]. Aphorpe demonstrated similar leakage for Android IM apps over Wi-Fi [?]. Recent machine-learning classifiers identify not only event types but also multimedia uploads and voice-over-IP handshakes in QUIC traffic [?]. Countermeasures such as adaptive padding raise bandwidth by 8–14 % while mitigating most classifiers [?].

(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 [?]. Kwon showed that Telegram’s “People Nearby” feature enables trilateration of user location with meter-level accuracy [?]. Signal introduced private-set-intersection with SGX enclaves to curb this vector, yet follow-up work revealed side-channels via cache timing [?].

(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 [?]. 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 [?]. Recent GDPR-oriented studies show that less than 40 % of messaging apps provide complete data-export facilities despite legal requirements [?].

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 Research Gap

Most prior work approaches Android behaviour analysis from one of three angles. *Static inspection* (e.g., code-review and manifest checks) is efficient but often misses behaviours that surface only at run time or behind obfuscation. *User-space monitors*—such as taint-tracking or framework hooks—offer richer semantics, yet typically require a modified or rooted runtime, can be noticed by well-crafted apps, and place implicit trust in native code. Finally, *kernel-level tracing* delivers low overhead and strong tamper resistance, but the raw syscalls and Binder op-codes it yields expose little intent, leaving a considerable semantic gap.

Taken together, these observations suggest there is still room for techniques that keep the light footprint and robustness of kernel tracing while adding just enough context to make the resulting signals useful for privacy and security reasoning—ideally without demanding special firmware or extensive system modifications.

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

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 appli-

cation 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.

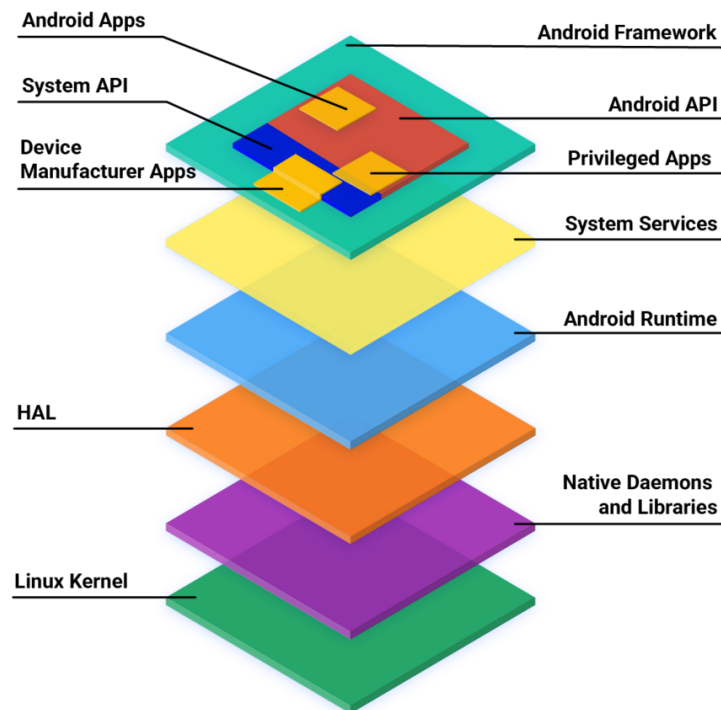


Figure 3.1: Updated Diagram of Android Software Stack (source: Android Developers Guide [?]).

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 sys-

tem 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 ?? . 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.

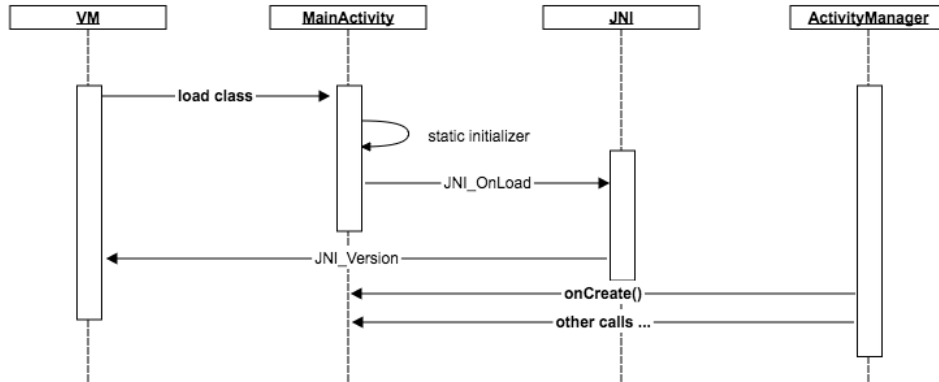


Figure 3.2: Sequence diagram showing the JNI initialization flow in an Android application.

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 [?]. 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.

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 hybrid methodology combining static and dynamic analysis to profile behavioral patterns of popular Android messaging applications without requiring app instrumentation.

Static analysis was employed as the initial step of the hybrid approach due to its ability to rapidly identify potential privacy-sensitive behaviors directly from application binaries without execution. By analyzing application code statically, it is possible to preemptively identify potential leakage points, risky APIs, and suspicious interactions that merit further dynamic investigation. Static analysis provides scala-

bility, repeatability, and early-stage detection capabilities, significantly reducing the scope and complexity required for dynamic analysis.

Androguard was chosen for primary static analysis due to its ease of use, rapid execution, and convenient Python-based interface, allowing seamless integration with existing scripting and processing workflows. Subsequently, Mobile Security Framework (MobSF)—an open-source security analysis platform that integrates both static and dynamic analysis capabilities—was utilized. MobSF was selected because of its comprehensive reporting features, intuitive graphical interface, and efficient analytical capabilities, streamlining the analytical process and facilitating deeper insights into application behavior.

The dynamic analysis leveraged kernel-level tracing, chosen for its transparency, accuracy, and non-invasive nature, relying heavily on existing capabilities provided by SYSDROID, which was extended further for the needs of this analysis. Experiments were conducted on a rooted Android device using Magisk and bootloader unlocking to access the kernel tracing filesystem (tracefs). Tools employed include ftrace, kprobes, ADB, custom shell scripts, and Python utilities for parsing and processing trace data into structured formats (JSON).

An essential design decision was the use of kprobes and ftrace instead of more advanced technologies such as eBPF, aligning with the principle of simplicity. Established tracing tools were sufficient to meet the research objectives, avoiding unnecessary complexity and facilitating ease of deployment, lower learning curve, and minimized potential system instability or compatibility issues.

Tracing was initiated using bash scripts executed directly on the Android device, dynamically configuring and activating kprobes and tracepoints targeting specific kernel events. Raw trace data from tracefs was exported securely to the host system, processed, and cleaned via Python scripts, employing PID-based filtering and heuristic techniques.

Design principles such as reproducibility, low overhead, and capability to detect IPC, sensitive resource access, and network activities were prioritized. A sliding window approach enabled detailed temporal analysis, ensuring data quality through validation and heuristic cleaning to filter irrelevant activities.

Visualization involved generating static summaries using Python libraries (e.g., matplotlib) for statistical evaluation and event distributions. Concurrently, an interactive web-based visualization tool, developed using Flask and D3.js, provided dynamic filtering and detailed timeline-based interaction summaries.

Detailed descriptions of the methodology and pipeline implementation are elaborated in subsequent chapters.

4.2 Static Analysis Pipeline

4.2.1 Tool Selection and Rationale

The static analysis phase relied primarily on two open-source tools: Androguard and MobSF. These tools were selected based on their usability, extensibility, and ability to produce structured, interpretable results suitable for early-phase exploration.

Androguard was chosen as the first-stage analyzer due to its lightweight nature and Python-based interface. It allows decompilation of APK files into Dalvik byte-code, extraction of manifest metadata, and traversal of call graphs. Its scripting capabilities enabled batch processing and direct generation of structured CSV reports. Specifically, the CSV file used in this study summarizes permission usage, class hierarchies, and API call references for quick inspection.

Following this, MobSF (Mobile Security Framework) was used for deeper inspection. MobSF is a versatile tool supporting both static and dynamic analysis through a unified web interface or API. It performs deep inspection of Android APKs and generates detailed PDF reports. In this study, MobSF’s output included permissions, dangerous API patterns, intent filters, and trackers, as demonstrated by the PDF report generated for the Telegram APK.

The combined use of these tools offered both programmatic flexibility and interpretability. Androguard served as the automated backend for pre-filtering and rapid metadata extraction, while MobSF provided comprehensive summaries that supported manual cross-verification and security-relevant insights. This dual-tool strategy enabled efficient triaging of application behavior prior to dynamic analysis.

4.2.2 Setup and Execution Workflow

The APKs of the three messaging applications (Signal, Messenger, Telegram) were extracted directly from the device using ADB. These APKs were then analyzed using both Androguard and MobSF in separate environments.

Androguard was set up in a Python virtual environment on WSL2. After installing version 4.1.3 and its dependencies, a custom script was used to automatically extract information about permissions, component exposure, cryptographic usage, suspicious API calls, and code structure. The output was exported in structured CSV and JSON format, offering a quick behavioral profile for each application.

MobSF, on the other hand, was run through Docker for convenience and reproducibility. The official Docker image was used to launch the web-based analysis engine, which provided high-level reports summarizing potentially dangerous permis-

sions, intent filters, trackers, and native libraries. Each PDF report enabled visual inspection and supported security triage.

Together, these tools offered a balanced combination of lightweight automated analysis and rich visual reporting, enabling a fast yet informative static assessment of each target application.

4.2.3 Challenges and Limitations

While Androguard and MobSF were successfully integrated into the pipeline, the broader landscape of static analysis tools for Android presented significant challenges. Several well-established tools proved difficult to use in practice. For example, FlowDroid required complex setup involving specific Java versions and Android platform SDKs, and frequently failed to resolve context-sensitive paths in larger applications. Similarly, tools like Amandroid and QARK either lacked active maintenance or had compatibility issues with recent APK formats and OS environments. In particular, several tools failed to handle APKs signed with modern signature schemes (v2/v3), or encountered class resolution errors due to obfuscated code and multidex structures. As a result, their integration was deemed impractical within the constraints of this study’s timeline and reproducibility requirements. This further justified the decision to rely on tools with minimal dependencies and verified stability, prioritizing efficiency over theoretical completeness.

4.3 Dynamic Analysis Pipeline

4.3.1 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.

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 ??.

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

Listing 4.1: Rooting commands for device setup

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.

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 ??.

```
1 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.

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 ??.

```
1 adb shell
2 for f in $(find /sys/class/ -name dev); do
3     echo "$f -> $(cat $f)"
4 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 ??.

```
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.

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 ??, demonstrating how each sensitive resource is uniquely identified by a combination of its inode, major/minor device numbers, and a descriptive label.

```
1 {
2     "contacts": {
3         "st_dev16": 65102,
4         "st_dev32": 266338382,
5         "major": 254,
6         "minor": 78,
```

```

7     "inode": 9253,
8     "path": "/data/data/com.android.providers.contacts/databases/contacts2.
      db",
9     "description": "Contacts database"
10  }
11 }

```

Listing 4.5: Serialized JSON metadata for sensitive resource mapping

4.3.2 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 ?? 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 ??.

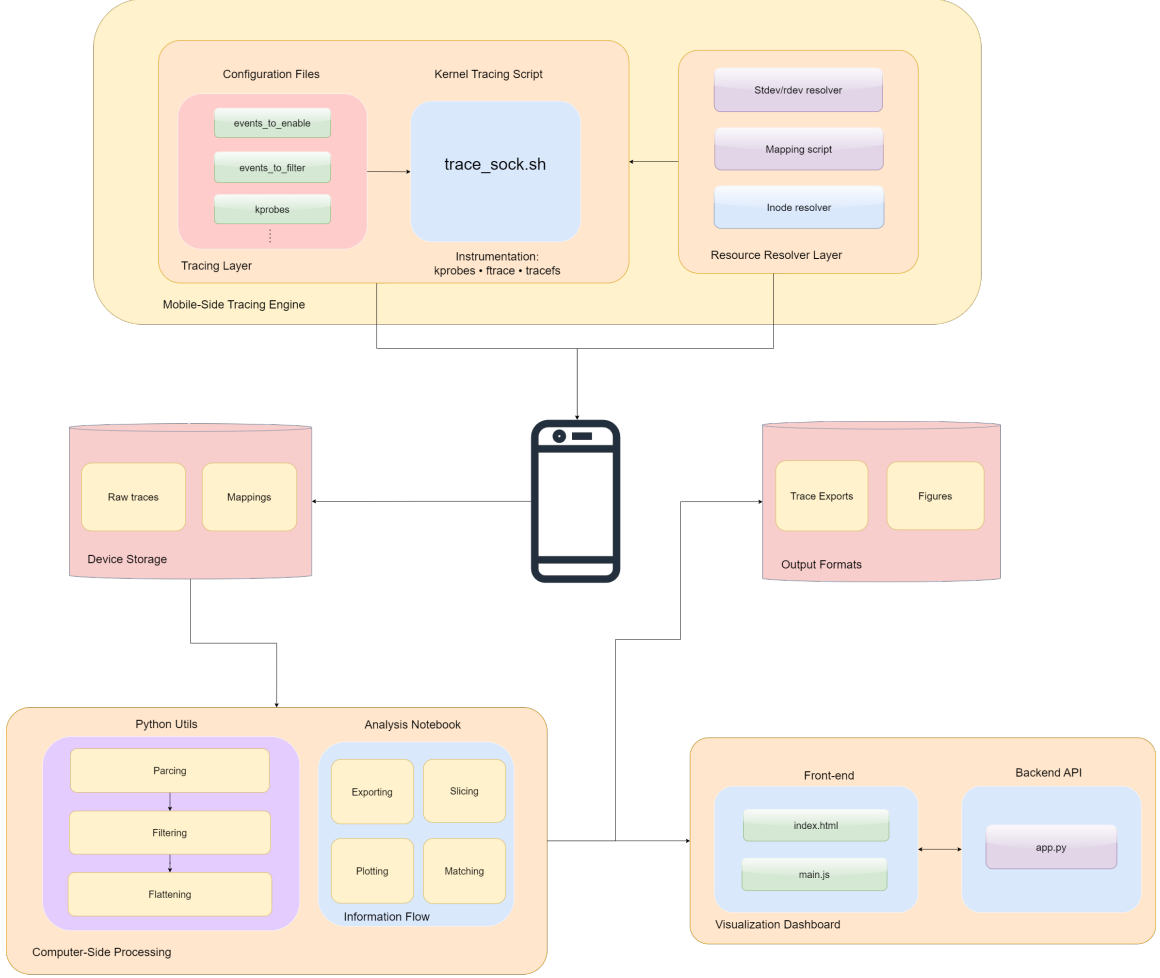


Figure 4.1: High-level modular architecture and data-flow pipeline of xSYSDROID.

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.

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:

```

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

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 $\{\text{category} \rightarrow [\text{identifiers}]\}$ where $\text{category} \in \{\text{camera}, \text{audio_in}, \text{bluetooth}, \text{gnss}, \text{nfc}, \text{contacts}, \text{sms}, \text{calllog}, \text{calendar}\}$. 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 ?? 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.

Parsing and Pre-Processing (Computer-Side)

After the raw kernel traces were collected from the device, they were quickly pre-processed using a lightweight Python parser. Each trace line was split using regular expressions to extract the main header fields (task, PID, CPU, flags, timestamp, event name) and the detailed key-value pairs. Numerical fields were converted to plain decimal form, flags were normalized, and paths were cleaned of extra characters.

Next, a sequence of basic filters removed background noise: well-known system daemons were excluded, short-lived file descriptors were ignored, and redundant Binder transaction chatter was collapsed. Finally, the cleaned trace events were divided into overlapping windows to enable manageable and context-preserving analysis in the next steps. This simplified approach followed the same principles as in SysDroid but was adapted for faster exploratory profiling.

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 $\{p\}$). $IPC \rightarrow$ 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; $IPC \rightarrow$ 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 $V = \text{PIDs}$ 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:

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

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.

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.

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.

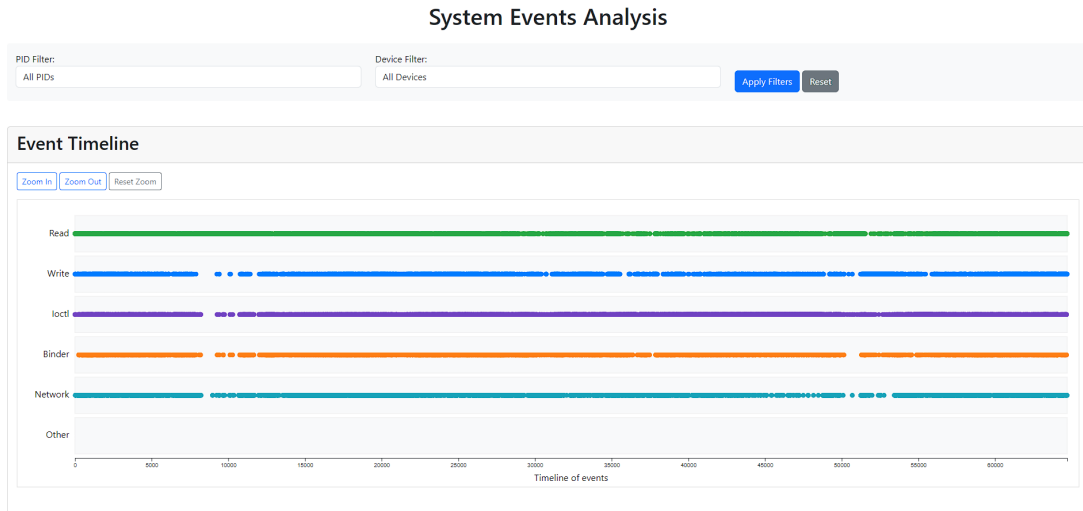


Figure 4.2: Interactive event timeline displaying categorized syscall activity over time. Users can zoom, filter by PID and device, and focus on specific syscall types.

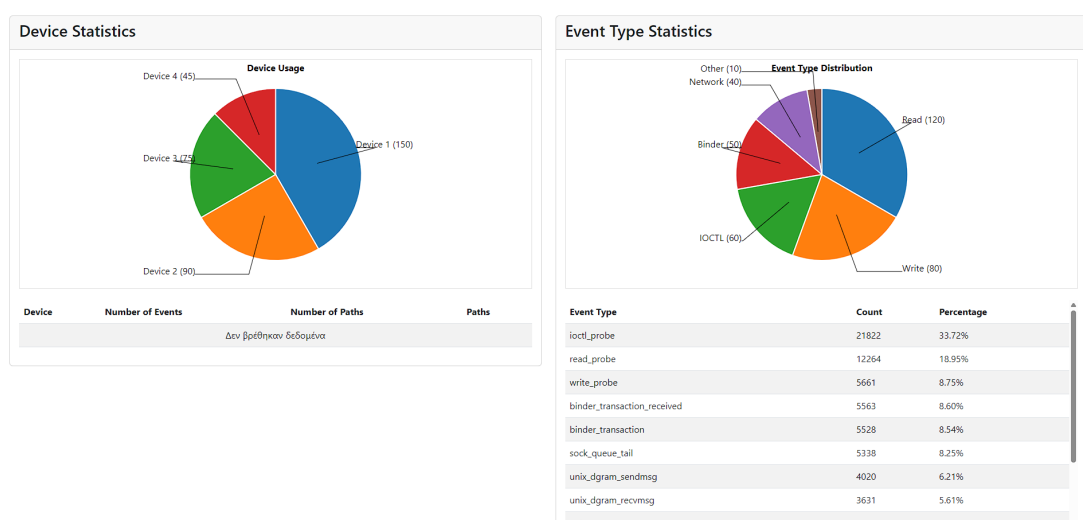


Figure 4.3: Aggregate statistics panel showing device usage and syscall distribution. The interface displays counts and percentages, updating dynamically as filters are applied.

Pipeline Summary

To close this architecture chapter, Figure ?? 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.

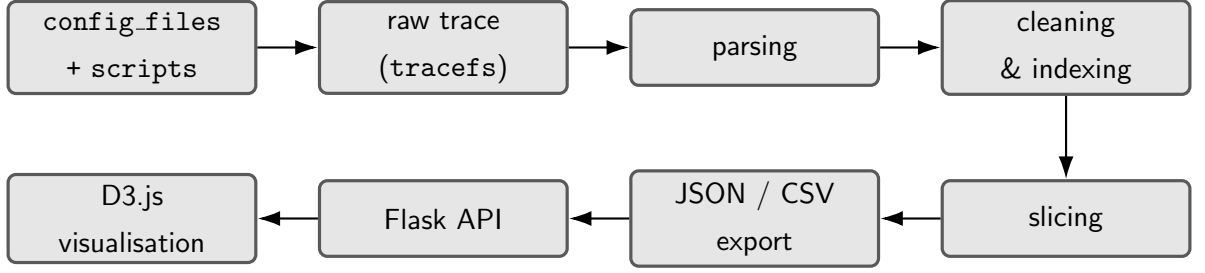


Figure 4.4: End-to-end data flow from probe configuration to interactive analytics.

4.4 Workload Design and Usage Scenarios

4.4.1 Target Application

Three representative Android messaging applications were selected: Signal, Messenger (Meta), and Telegram. Signal stands out for its open-source Signal Protocol, which enforces strict E2EE and minimal metadata, making it a strong benchmark for privacy behavior. Messenger, as a dominant commercial product, provides high-volume, complex user interaction patterns. Telegram adds diversity with its cloud-based and secret chat modes and public channels. This focused corpus balances privacy strength, usage scale, and technical feasibility.

4.4.2 Execution Scenarios and Scheduling

Dynamic tracing was executed under four systematically defined scenarios per app: full permissions, restricted permissions, background passive state, and active foreground use including messaging and video calls. Each run lasted around five minutes and was repeated to ensure consistency. Static scans used the prepared scripts and tools to generate CSV, JSON, and PDF reports per APK, cross-validated with run-time findings.

4.4.3 Data Volume and Collected Artifacts

Static analysis produced compact artifacts (CSV ≤ 100 kB, PDF 100–200 kB). Dynamic traces were larger, typically 50–300 MB per session depending on activity and hook density. Table ?? summarizes static output sizes; dynamic logs provided high-resolution behavioral records for flow reconstruction and visualization.

Application	CSV size (kB)	PDF size (kB)
Signal	50	120
Messenger	80	180
Telegram	45	110

Table 4.1: Approximate static analysis file sizes

Chapter 5

Results

5.1 Static Analysis Results

This chapter provides a comprehensive *static* security evaluation of three prominent messaging applications—*Signal*, *Telegram*, and *Meta Messenger*—based on production-signed APKs sourced from the Google Play Store in June 2025. The analysis leverages advanced tools including MOBSF, ANDROGUARD, and APKID to thoroughly inspect embedded resources, native libraries, and signing certificates.

5.1.1 Scope and Methodology

The static assessment was carried out in five successive stages. **Manifest Analysis** reviewed sensitive flags—most notably `allowBackup`, the set of exported components, and every declared permission. **Bytecode Inspection** then examined the compiled `.dex` payloads, hunting for unsafe coding practices and known vulnerability patterns. **Native Library Hardening** evaluated bundled native libraries against modern defence mechanisms, checking for NX, PIE, stack canaries, RELRO and FORTIFY usage. **Secret Scanning** applied entropy-based heuristics to surface hard-coded secrets such as API keys or credentials. Finally, **Anti-Analysis and Obfuscation** mapped anti-debug measures, anti-VM tricks and compiler artefacts to gauge each app’s resistance to reverse engineering.

5.1.2 Detailed Application Findings

The following pages present the most relevant static observations for the three evaluated messaging apps. We begin with basic APK metadata, continue with a high-level view of exposed components, and close with the severity distribution reported by MOBSF. The discussion is intentionally brief and focuses on facts that are likely to

affect maintenance effort or security posture.

APK Metadata Overview

All three applications already target Android 14 (API 34) or newer, with **Messenger** going a step further to API 35—an advantage for users on the bleeding-edge OS track. The declared minimum SDK, however, differs noticeably: **Signal** still supports devices back to API 21 (Android 5.0), **Telegram** to API 23 (Android 6.0), whereas **Messenger** drops legacy versions entirely by setting its floor at API 28 (Android 9). Shrinking that compatibility window simplifies maintenance and removes long-standing platform bugs, though at the expense of excluding pre-Pie users.

Package size presents an intriguing anomaly. Although **Messenger** contains by far the most extensive component inventory—nearly five-hundred activities and over one-hundred services—its APK is only 66MB, markedly smaller than Signal’s 86MB. Differences in compression techniques, split-APK delivery for native libraries, and stronger reliance on on-demand feature modules likely account for part of this gap. The apparent mismatch between component complexity and binary footprint will be revisited in the subsequent analysis.

Table 5.1: Key APK Metadata (June 2025)

App	Min SDK	Target SDK	APK Size
Signal	21	34	85.6 MB
Telegram	23	34	43.2 MB
Messenger	28	35	66.3 MB

Static Findings Severity Overview

MobSF’s aggregate grading shows that all three applications sit in the same “B-range,” yet their internal risk profiles diverge markedly. **Signal** records the fewest high-severity issues (three), suggesting a comparatively tight baseline configuration, although a non-negligible volume of medium findings (forty-three) still demands sustained remediation effort. **Telegram** doubles the high-risk tally and edges ahead in medium alerts, indicating broader exposure despite its leaner component graph. **Messenger** stands out: ten high-severity items and almost a hundred medium-level flags underscore the operational complexity already noted in its manifest. While the platform does earn one additional “Secure” mark, the ratio of critical to informational findings highlights a heavier security debt that subsequent sections will examine in detail.

Table 5.2: Findings Severity (MobSF)

Severity Level	Signal	Telegram	Messenger
High	3	5	10
Medium	43	47	98
Info	3	4	3
Secure	3	3	4
Hotspot	2	1	1

Component Analysis

Among the three applications, **Messenger** exhibits by far the greatest architectural complexity, bundling close to five hundred activities together with more than one hundred background services; this breadth of functionality inevitably widens the attack surface and imposes a substantial testing burden. By contrast, **Telegram** maintains the smallest public activity surface yet still exposes fifteen services—roughly twice the number found in **Signal**—so careful permission-guarding is essential to mitigate potential component-hijacking risks. **Signal**, meanwhile, publishes no content providers at all, thereby curtailing direct inter-application data exchange, although its nineteen exported activities remain non-trivial and should be reviewed on a recurring basis.

Table 5.3: Detailed Comparison of App Components

Component Type	Signal	Telegram	Messenger
Activities	89	16	496
Services	25	28	114
Receivers	30	25	77
Exported Activities	19	14	12
Exported Services	7	15	25
Exported Receivers	4	3	14
Exported Providers	0	1	6

Manifest posture

As shown in ??, the MobSF scan exposes three distinct risk profiles. Telegram leads with *two High*-severity findings and thirty-six warnings, signalling several manifest settings that could be exploited if left unpatched. Signal reports a single *High* issue plus thirty-two warnings—evidence of a somewhat tighter, though still imperfect, configuration. Messenger has no high-severity items, yet its eighty-six warnings reveal a “many-small” pattern of misconfigurations that can accumulate technical debt, especially given Meta’s rapid release cadence.

Turning to the individual flags in ??, all three applications keep `allowBackup=true`, meaning a device in developer mode can exfiltrate private app data unless

Table 5.4: Manifest Issues by Severity (MobSF)

Severity Level	Signal	Telegram	Messenger
High	1	2	0
Warning	32	36	86
Info	0	0	0

users disable OEM backup routes. Only Telegram and Messenger permit clear-text HTTP traffic globally—a policy that undermines transport-layer confidentiality and should be restricted to explicit, whitelisted domains. Finally, Messenger presents the largest externally visible surface with forty-eight exported components, followed by Telegram at thirty-three and Signal at nineteen; every exported entry must be permission-guarded or otherwise verified to avoid component-hijacking attacks.

Table 5.5: Critical Manifest Flags

Flag	Signal	Telegram	Messenger
allowBackup	✓	✓	✓
cleartextTraffic	—	✓	✓
exportedComponents	19	33	48

Permission context

Requesting “dangerous” permissions is—at least to a point—inevitable for modern IM clients: contact synchronisation (`READ_CONTACTS`, `WRITE_CONTACTS`) underpins address-book integration, storage permissions allow media exchange, while microphone, camera, and location access enable voice messages, video calls, and live-location sharing. These rights, however, also classify as *high-risk vectors* because they unlock direct access to PII, sensors, and the filesystem. Notably, **Telegram** and **Messenger** extend their reach with system-level privileges such as `CALL_PHONE`, `SYSTEM_ALERT_WINDOW`, and full account management—features that power in-app calling, chat-head overlays, or multiple account workflows, but simultaneously broaden the abuse surface (e.g. overlay phishing or component hijacking). **Signal**, by contrast, adopts a *minimal-necessary* stance, omitting phone-call control, overlay windows, background location, calendar access, and package-installation rights. This leaner footprint limits the attack surface without materially impairing core functionality, illustrating a tighter alignment with the principle of least privilege.

Bytecode-level exposure. Static inspection of the `.dex` payloads (??) confirms a common baseline of cryptographic missteps—most notably the continued use of AES in CBC mode without accompanying integrity checks. Both **Messenger** and

Table 5.6: Differential Dangerous Permissions (Top 10)

Permission	Signal	Telegram	Messenger
USE_CREDENTIALS	✓	—	—
MANAGE_ACCOUNTS	—	✓	✓
SYSTEM_ALERT_WINDOW	—	✓	✓
CALL_PHONE	—	✓	✓
REQUEST_INSTALL_PACKAGES	—	✓	—
ACCESS_BACKGROUND_LOCATION	—	✓	—
READ_CALL_LOG	—	✓	—
READ_CALENDAR	—	—	✓
WRITE_CALENDAR	—	—	✓
BLUETOOTH_CONNECT	—	✓	✓

Telegram further embed debug-enabled **WebViews**, a design choice that simplifies troubleshooting but risks remote code-execution should those views become reachable at run time. **Signal** avoids that trap yet implements a permissive “trust-all” SSL fallback, undermining certificate validation in edge cases. World-writable artefacts are exclusive to Messenger, compounding the impact of its larger secret corpus.

All three binaries leak strings that score above the Shannon-entropy threshold typically used to flag embedded keys or tokens. Although some hits are likely innocuous (e.g. public RSA moduli or test vectors), the order-of-magnitude difference is revealing: **Signal** contains roughly 492 such literals, **Messenger** about 372, whereas **Telegram** exposes only 23. A dense concentration of opaque byte arrays suggests in-app storage of encryption keys, API credentials, or feature flags that could be harvested by an attacker with static-analysis skills. Even if individual values prove non-sensitive, their sheer volume complicates code review and widens the search space for vulnerability researchers.

Taken together, while all three projects inherit medium-tier warnings—raw SQL strings, legacy digests, and non-cryptographic PRNGs—the distribution of high-severity items continues to underscore a stricter secure-coding discipline in Signal when compared to its competitors, albeit with the caveat of its unusually large pool of embedded high-entropy data.

Third-party telemetry vs. privileged scope. ?? shows that only **Messenger** bundles external analytics or debugging SDKs (*Flipper*, Google Analytics, Mapbox). While these libraries facilitate crash reporting and in-app mapping, they also widen the app’s network footprint and introduce code that is not fully controlled by the first-party vendor. **Telegram** and **Signal**, by contrast, ship with no declared tracker packages, matching their privacy-first messaging.

Table 5.7: Comparative Matrix of Code-Level Vulnerabilities

Issue	Messenger	Telegram	Signal
High-severity findings	4	5	3
Remote-debuggable <code>WebView</code>	✓	✓	✗
AES-CBC without integrity	✓	✓	✓
Trust-all SSL trust manager	✗	✗	✓
World-writable/readable files	✓	✗	✗
Raw SQL injection risks	Warning	Warning	Warning
Weak digests (MD5, SHA-1)	Warning	Warning	Warning
Insecure PRNG usage	Warning	Warning	Warning
Hard-coded high-entropy secrets	372	23	492

Risk, however, is not dictated by trackers alone. ?? intersects each manifest with a canonical set of 25 Android permissions that mobile-malware signatures most frequently abuse (compiled from Koodous and VirusTotal rule bases). Telegram requests 18/25 of these privileges, slightly more than Messenger (16/25) and clearly above Signal (15/25). Many of the flagged permissions (`CAMERA`, `RECORD_AUDIO`, fine location) are legitimate for a modern messaging suite; nonetheless, they enlarge the attack surface an adversary could exploit after a successful code-execution bug. Signal therefore retains the leanest privilege envelope while also avoiding embedded trackers.

Table 5.8: Declared third-party tracker SDKs

App	Embedded SDKs detected by MobSF
Messenger	Facebook Flipper, Google Analytics, Mapbox
Telegram	—
Signal	—

Table 5.9: Overlap with 25 malware-associated permissions

App	Dangerous permissions requested
Signal	15/25
Telegram	18/25
Messenger	16/25

Firestore back-end exposure. Messenger diverges architecturally from its rivals: instead of relying on third-party back-ends it maintains a proprietary service mesh—historically rooted in custom XMPP extensions and, more recently, a gRPC-based “MqttLite” transport—so no Firestore endpoints appear in the binary. Static string inspection, on the other hand, reveals hard-coded Firestore Realtime Database

URLs in both Signal and Telegram (??). MobSF flags these only as *info* because the associated Remote-Config queries are refused (HTTP 4/403), implying that anonymous feature-flag pulls are disabled. That safeguard hinges on server-side rules: if at any point permissive read/write ACLs are introduced—intentionally or by accident—the same endpoints could expose chat metadata or configuration secrets and escalate from “low noise” to a critical data-leak vector.

5.1.3 Cross-Application Comparative Synthesis

Static inspection assigns all three apps a *B-tier* MobSF grade, yet the underlying risk landscape diverges sharply. **Messenger** emerges as the most exposed candidate: clear-text traffic remains enabled, three telemetry SDKs are bundled, and 372 hard-coded secrets coexist with four high-severity code findings. **Telegram** fares marginally better—no trackers and only 23 secrets—but it still requests the widest set of malware-aligned permissions and carries five top-rank issues. **Signal** presents the leanest attack surface overall (no trackers, strict permission profile), although its secret sprawl (~500 literals) and a “trust-all” SSL fallback warrant immediate attention.

Table 5.10: Aggregate static-risk indicators.

App	MobSF Score	High	Secrets	Trackers
Messenger	48 (B)	4	372	3
Telegram	49 (B)	5	23	0
Signal	51 (B)	3	492	0

Brand prominence on its own is no proxy for security maturity. At the same time, code-base size and the age of the original release carry weight: the larger and older an application, the more legacy modules and technical debt it is likely to accumulate. Messenger—by far the heaviest and oldest code-base in the set—consequently lags behind Telegram and the leaner, newer Signal in several hardening metrics. Regardless of pedigree, however, all three projects need continuous dynamic testing and refactoring to curb the security drag that inevitably accompanies growth over time.

5.2 Dynamic Analysis Results

5.2.1 Scope and Dataset Overview

5.2.2 Behavioural Patterns Observed

5.2.3 Statistical Summaries

5.2.4 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](https://github.com/jacquesklein2302)
- [41] Oteau 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., Oteau 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., Kaffle K., Moran K., Nadkarni A., Poshyvanyk D. Discovering flaws in security-focused static analysis tools for Android using systematic mutation. *USENIX Security*, 2018. [arXiv](https://arxiv.org)

- [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 Organizations*. 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.