Athens University of Economics and Business

Department of Management Science and Technology

Undergraduate Thesis

The Kernel Never Lies: A Hybrid Security Analysis of Evasive Behaviors in Messaging Apps

Student:

Foivos - Timotheos Proestakis

Student ID: 8210126

Supervisors:

Prof. Diomidis Spinellis

Dr. Nikolaos Alexopoulos

Submission Date:

June 30, 2025

Abstract

This thesis introduces a hybrid security analysis methodology for generating detailed behavioral profiles of popular mobile messaging applications by combining static inspection with custom kernel-level dynamic tracing. Specifically, the dynamic tracing extends the SliceDroid framework with a custom web dashboard, enabling visualization of kernel-level events such as system calls, IPC, and network activity.

To demonstrate its effectiveness, comprehensive static and dynamic analyses were performed on three widely used messaging apps: Signal, Telegram, and Meta Messenger. The static inspection, conducted with tools such as MobSF, APKID, and Androguard, highlighted distinct differences in manifest configurations, permission scopes, cryptographic implementations, and third-party integrations — notably, Messenger was found to request and utilize broader contact and storage permissions compared to the more privacy-restrictive Signal.

For dynamic evaluation, the extended SliceDroid collected precise runtime data under realistic usage scenarios, capturing file system operations, IPC calls and network flows. The combined analyses uncovered significant behavioral differences: Messenger exhibited the highest runtime complexity and heavy IPC usage, and notably, it was observed accessing and synchronizing user contacts even when explicit contact permissions were revoked, indicating a potential violation of user privacy that static analysis alone could not reveal.

Overall, these findings highlight the value of integrating static analysis with kernel-level tracing to reveal hidden operations and inform more robust security and privacy evaluations of mobile applications.

Acknowledgments

I would like to express my gratitude to Dr. Nikolaos Alexopoulos, for his invaluable guidance, insightful feedback, and continuous support throughout the duration of this thesis. His 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.

[June 30, 2025] [Foivos - Timotheos Proestakis]

Contents

List of Acronyms

ADB Android Debug Bridge

AMS ActivityManagerService

ANOVA Analysis of Variance

AOT Ahead-Of-Time

API Application Programming Interface

APK Android Package Kit

ART Android Runtime

DBI Dynamic Binary Instrumentation

DEX Dalvik Executable

E2EE End-to-End Encryption

eBPF extended Berkeley Packet Filter

FCM Firebase Cloud Messaging

GDPR General Data Protection Regulation

GID Group ID

GKI Generic Kernel Image

HAL Hardware Abstraction Layer

HSD Honestly Significant Difference

ICC Inter-Component Communication

IPC Inter-Process Communication

JIT Just-In-Time

JNI Java Native Interface

JSON JavaScript Object Notation

MAC Mandatory Access Control

MobSF Mobile Security Framework

NX Non-Executable Stack

OTA Over-The-Air

PIE Position-Independent Executable

PII Personally Identifiable Information

PMS PackageManagerService

PRNG Pseudo-Random Number Generator

RELRO Relocation Read-Only

SDK Software Development Kit

SELinux Security-Enhanced Linux

SQL Structured Query Language

SSL Secure Sockets Layer

TCP Transmission Control Protocol

TLS Transport Layer Security

UDP User Datagram Protocol

UID User ID

UI User Interface

WAL Write-Ahead Logging

WSL Windows Subsystem for Linux

XMPP Extensible Messaging and Presence Protocol

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 [50]. Among the dominant mobile platforms, Android—an open-source operating system developed by Google—holds a stable global market share of approximately 75% [56]. 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 Telegram, 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 [48], [39].

Incidents such as Facebook's unauthorized collection of SMS texts and call logs from Android devices [48] 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 [82, 83]. 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 [9].

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 [29, 10, 62]. Messaging apps may still leak information through patterns such as message timing or background system activity—even when message payloads are encrypted.

Addressing these challenges requires a comprehensive understanding of how messaging apps are structured and how they behave at runtime. This thesis adopts a hybrid approach that combines static analysis—examining app manifests, permissions, and embedded components—with kernel-level tracing of live system calls to capture detailed behavioral evidence [6, 70]. Kernel-level tracing is a powerful technique for runtime visibility: tools such as `ftrace` [78] and `kprobes` [79] record system-call activity with minimal overhead, offering fine-grained insights into how an application interacts with operating system resources . Despite its potential, only a handful of studies have systematically applied this method to messaging apps [40], despite their widespread and sensitive use—including by government and military officials [52].

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 [48]. 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.[48, 54, 52].

Problem Statement Although previous work has addressed various facets of Android application security and behavior, comprehensive comparative studies of privacy-sensitive messaging apps that combine static security inspection with systematic kernel-level tracing remain limited [13]. As a result, critical aspects of both declared configurations (e.g., permissions, exported components) and actual runtime behavior—including low-level system interactions—are still insufficiently understood. This study seeks to close this gap by integrating static analysis with dynamic kernel-level tracing under realistic usage scenarios, thereby supporting more reliable, transparent assessments of messaging app behavior and informing future security research.

1.2 Research Objectives

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

Primary Objective

To devise a robust, hybrid tracing methodology that uncovers data-minimisation breaches, permission–runtime mismatches, and hidden data access, while contrasting privacy-centric messengers (e.g., *Signal*) with mainstream platforms.

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.
- Correlate static findings with dynamic behavior to highlight security and privacy risks.
- 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 systematic methodology can be used for a comparative security and privacy analysis of mobile messaging applications?

Q2. Is *static* analysis alone sufficient to uncover privacy-intrusive behaviour, or do kernel-level traces reveal additional hidden operations and syscall patterns?

Q3. Which system calls do popular messaging apps issue during normal use, and how do these calls align with—or diverge from—their declared permissions?

Q4. How do privacy-centric apps (e.g., *Signal*) compare to mainstream platforms in terms of data minimisation, syscall footprint, and overall privacy posture?

1.4 Limitations of the Study

To provide a realistic perspective on the scope of this thesis, it is important to distinguish between constraints that arise from project-specific conditions and those inherent to kernel-level behavioural analysis and static inspection.

Project-Specific Constraints.

- *Timeframe:* The work was conducted within the timeframe of an undergraduate thesis, which constrained the number of test iterations, devices, and workload scenarios. For instance, only text messaging, voice clips, and image sharing were evaluated dynamically, while video calls—which would likely introduce significant additional real-time UDP streams—were not included.
- *Hardware Diversity:* Runtime tracing was performed on a limited set of consumer-grade Android devices. Results may not generalise to tablets, devices with custom ROMs, or older low-end hardware where kernel instrumentation behaves differently or is partially restricted.
- *App Selection:* The analysis covered only three prominent instant messaging clients (Messenger, Telegram, Signal). Other messaging or region-specific apps, as well as system-bundled communication tools, were excluded, potentially leaving out unique interaction or security patterns.
- *Infrastructure Limitations:* Local storage and compute capacity limited the duration and granularity of traces. High-frequency events or prolonged sessions could not be captured exhaustively without incurring unsustainable storage overhead.

Domain and Methodological Constraints.

- *Package Scope:* The mapping algorithm reliably lists user-installed APKs but excludes protected system apps like Chrome.
- *Trace Volume:* Continuous tracing generates large raw data with redundant or low-entropy segments; filters and time limits reduce size but may omit rare behaviors.
- *Attribution Noise:* Shared system services (e.g., `servicemanager`) introduce IPC noise, causing possible cross-app misattribution.
- *Network Depth:* The network layer analysis focused on event-level capture without deep packet inspection or full protocol parsing.

Overall, these limitations do not compromise the core findings but should be kept in mind when interpreting and generalising the results.

1.5 Contributions of this Thesis

This thesis makes the following key contributions to the field of mobile security and privacy analysis:

i. Proposes and implements an enhanced kernel-level tracing methodology, building upon the SliceDroid framework, enabling precise monitoring of mobile application runtime behavior.

ii. Provides a systematic hybrid analysis approach combining static and dynamic methods, resulting in comprehensive behavioral profiles for security-critical applications.

iii. Develops and integrates a custom package-to-process name mapper, allowing accurate identification and attribution of runtime events to the corresponding commercial application, even when system-level identifiers differ.

iv. Delivers an in-depth comparative analysis of three major messaging applications (Signal, Telegram, Meta Messenger), highlighting significant architectural and operational differences impacting user privacy.

v. Statistically validates observed differences through rigorous inferential testing (ANOVA and Tukey HSD), offering robust, evidence-based conclusions.

vi. Identifies critical privacy-related insights, including permission discrepancies and hidden runtime behaviors.

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 the limitations of Android permissions and E2EE, motivates the combined static and dynamic approach, and states the thesis goals, research questions, and scope.

Chapter 2 – Related Work: Reviews static security analysis, user-space and kernel-space dynamic instrumentation, and prior privacy studies on messaging apps, pinpointing the gap in systematic inspection of declared configurations alongside runtime profiling.

Chapter 3 – Technical Background: Provides an overview of the Android architecture relevant to this study, introduces key system components and security features, and describes characteristics of the selected messaging applications.

Chapter 4 – Methodology & System Design: Describes the overall approach, detailing two complementary pipelines: one for static inspection and another for dynamic tracing.

Chapter 5 – Results: Presents the outcomes derived from both pipelines: static findings such as manifest structures, permission declarations, and embedded third-party modules, alongside dynamic measurements of syscall distributions, permission–behavior mismatches, and comparative visual summaries of privacy-relevant runtime patterns.

Chapter 6 – Discussion: Relates results to the research questions, and outlines methodological limitations and suggested improvements.

Chapter 7 – Conclusions: Recaps contributions, emphasises the value of combining static inspection with kernel-level tracing for privacy audits.

Appendix A – Source Code: Key implementation scripts supporting the App Mapper functionality.

Chapter 2

Related Work

The purpose of this chapter is twofold: (i) to review existing Android application analysis approaches, with a focus on kernel-level tracing methods, 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 and manifest offline, 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, research in this area has evolved significantly, moving from foundational permission analysis to sophisticated data-flow tracking and machine learning models to address challenges like inter-component communication and code obfuscation.

i) Permission and Manifest Analysis

Initial research focused on understanding the Android permission model itself. Foundational work by Felt et al. provided a deep dive into how permissions are declared in the manifest, requested by applications, and interpreted (or misinterpreted) by users [17]. These early studies established that declared permissions alone are often insufficient to predict malicious behavior, motivating the need for deeper code analysis.

ii) Privacy-Oriented Taint Analysis

To understand data misuse, research shifted to tracking information flows from sensitive *sources* (e.g., contact lists) to security-critical *sinks* (e.g., network sockets). **FlowDroid** set a new standard by introducing context-, flow-, field-, and object-sensitivity alongside an accurate activity lifecycle model, dramatically reducing false positives [3]. On the *DroidBench* benchmark suite, it achieved 93% recall and 86% precision, outperforming contemporary tools and scaling to large, real-world applications.

iii) Inter-Component Communication (ICC) Analysis

A primary limitation of early taint analysis was its inability to track data flows across process boundaries. To address this, **IccTA** augmented FlowDroid with an explicit ICC model, accurately mapping data passed through Intents and other IPC mechanisms [23]. It surpassed earlier tools like **Epicc** [35] and **CHEX** [15] on the *ICC-Bench* suite. **Amandroid** extended this concept by constructing a full inter-component data-flow graph, uncovering previously unknown vulnerabilities in hundreds of Play Store apps [44].

iv) Resilience to Obfuscation

As developers began using techniques like reflection, dynamic code loading, and native libraries to obscure behavior, static analysis tools had to adapt. **DroidRA** was designed to resolve reflective calls by using constant propagation to rewrite them into direct invocations, boosting FlowDroid's precision on reflection-heavy code by up to 60% [24]. Other approaches like **DexLEGO** reassemble bytecode captured at runtime for static analysis [32], while **LibDroid** specifically focuses on summarizing taint propagation within native C/C++ libraries [25], progressively widening the statically visible attack surface.

v) Learning-Based Malware Detection

A separate research strand moved beyond handcrafted rules towards machine learning. **Drebin** pioneered this by extracting lightweight syntactic features (permissions, API calls, etc.) from the manifest and bytecode to train a linear SVM classifier. It detected malware with 94% accuracy on a large dataset of 5,560 samples [2]. More recently, deep learning models like **DL-AMDet** have employed architectures combining CNNs and LSTMs to achieve near-perfect accuracy on public datasets [31], though their limited interpretability remains a challenge for high-assurance systems.

vi) Practical Analysis Toolkits

This body of research has culminated in powerful, extensible toolkits for practical security auditing. **MobSF** (*Mobile Security Framework*) integrates static and dynamic analysis, reverse engineering, and malware scoring in a user-friendly web interface [67], while **Androguard**, a Python-based library, provides the foundational components for building custom reverse engineering and analysis pipelines [63]. This thesis leverages both of these foundational tools: MobSF is employed for initial, broad-spectrum security scanning, while Androguard provides the programmatic basis for the custom static analysis scripts detailed in Chapter 4.

Synthesis. Static analysis has achieved substantial advances in precision and scale, evolving from simple manifest checks to sophisticated data-flow and machine learning models. Yet, it remains fundamentally constrained by its inability to observe runtime-specific behavior, implicit data flows, or dynamically loaded code. These blind spots, especially critical in privacy contexts, motivated the development of dynamic and hybrid approaches that can capture an application's *actual* behavior under real-world 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, and UI callbacks—without requiring a custom kernel. The literature has evolved along several complementary strands that echo, in spirit, the structure reviewed for static analysis.

(i) System-Call Monitors

Early approaches used the `ptrace` mechanism to intercept system calls externally. **DroidTrace**, for example, hooks every `ptrace` event to log the full system-call stream and can *force* rare code branches to execute, increasing coverage [47]. A later study demonstrated that it is possible to reconstruct high-level Android API invocations purely from syscall sequences, highlighting the semantic power of these low-level traces [33]. While these tools deploy on stock ROMs, they incur measurable overhead and cannot observe internal Java-level logic.

(ii) Dynamic Binary Instrumentation (DBI)

For analyzing native code, instruction-granular DBI engines like **Pin** (ported to ARM) enable fine-grained tracing, but suffer from high runtime cost and outdated Android support. More modern frameworks like **QBDI** improve portability to ARM/ Thumb-2 and withstand common anti-analysis techniques, although they remain native-only and typically require root privileges [53].

(iii) ART-Level Hooking Frameworks

A powerful category of tools modifies the Android Runtime itself to intercept method calls across the entire system. The most prominent example is the **Xposed Framework**, which alters the Zygote process to load custom modules at startup. This allows developers to hook, modify, and replace any Java method in any application, offering unparalleled control over app behavior [69]. While extremely powerful, this approach requires root access and system modification, which can break security features like SafetyNet and make it unsuitable for analyzing unmodified consumer devices.

(iv) In-Process Hooking Frameworks

In contrast to system-wide modification, tools like **Frida** inject a loader into a specific target process at runtime. This loader JIT-compiles JavaScript hooks into native code, permitting live patching of Java, JNI, and native functions for interactive experiments [65]. **DYNAMO** builds upon Frida to perform systematic differential analysis of framework evolution across Android versions [11], while **InviSeal** hardens Frida against detection and reduces its overhead to less than 3% [20]. However, these tools struggle with tamper-aware apps and are confined to a single process, leaving cross-app and kernel-level events hidden.

(v) Sandboxes and Emulators

To automate analysis at scale, containerised sandboxes like **C-Android** isolate apps within Linux namespaces for reproducible testing [7]. Emulation-centred pipelines (e.g., **DynaLog**) automate malware triage by collecting runtime features in parallel VMs [12]. Although high-throughput, these environments often diverge from real-device hardware and timing, and they are easily fingerprinted by sophisticated malware.

Synthesis. User-space instrumentation shines in its *semantic richness*, rapid deployment, and interactive control, making it ideal for annotating a 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. 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 our kernel monitors.

2.3 Kernel-Level Tracing Infrastructures

Tracing infrastructures at the kernel level offer a privileged vantage point beneath the Android framework, the ART runtime, and native user-space libraries. By instrumenting key OS subsystems—such as system calls, Binder IPC, and network interfaces—they expose low-level behavioral signals while incurring minimal runtime overhead. These signals are especially critical when applications encrypt payloads or obfuscate logic, rendering user-space instrumentation ineffective.

These infrastructures are built upon foundational Linux kernel mechanisms. In contrast to high-overhead, process-level tracers like `strace` [68], modern approaches leverage highly efficient in-kernel frameworks. These include `ftrace` [78], the kernel's native function tracer; `kprobes` [79] for dynamic function instrumentation; and `eBPF` [64], which allows for safe, programmable tracing directly within the kernel. The literature applying these tools can be grouped as follows:

(i) eBPF- and ftrace-Based Collectors

Modern frameworks such as **BPFroid** leverage eBPF and ftrace to capture fine-grained telemetry from stock Android kernels without requiring firmware modifications [6]. BPFroid monitors per-UID syscall sequences, Binder transactions, and network activities, enabling malware detection with low system overhead ($\sim$3% energy cost). Earlier efforts, such as **ID-Syscall**, used loadable kernel modules on Android 4.x to demonstrate that syscall n-grams alone can suffice for binary classification of app behavior [38], though such systems lacked semantic context and cross-process correlation.

(ii) Network-Centric and Covert-Channel Tracing

Beyond explicit syscall logs, more covert behavioral signals have also been explored. Celik and Gligor extract timing-based features from standard ftrace logs to detect stegomalware, highlighting how packet sizes and inter-packet gaps reveal anomalous communication even without payload inspection [8]. These techniques operate under

encryption but often omit Binder IPC, which is central to Android's inter-component messaging model.

(iii) Hardware-Assisted Monitors

Low-level performance monitoring units and instruction tracing extensions like ARM's Embedded Trace Macrocell (ETM) provide hardware-level insights with near-zero perturbation. For example, **HART** captures execution traces from proprietary kernel modules, offering unmatched visibility [46]. However, such approaches face practical limitations: ETM is often disabled in production smartphones, lacks mapping to user-level processes, and requires privileged access, constraining its applicability.

(iv) Production-Scale Tracing

Industrial-grade pipelines such as Google's **Perfetto** aggregate diverse sources, including ftrace and Binder logs—into rich, SQL-queryable trace files for profiling performance metrics like startup time and UI latency [51]. While primarily for performance diagnostics, such infrastructures are underutilized for security research. Complementary systems like **eMook** extend Perfetto by integrating per-PID eBPF filters, reducing overhead by 85% and enabling longitudinal field studies [45].

Synthesis. Kernel tracing offers a hard-to-tamper, always-on lens but must still tame five hurdles: the *semantic gap, opaque data, privilege requirements, data volume, and detectability*. Recent work narrows these by pairing syscalls with Binder provenance, granting BPF rights via SELinux instead of root, and summarising high-rate events in-kernel—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 [42]. 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 [30, 34]. While modern OS versions encrypt local backups, notification side-channels (e.g., Firebase tokens) remain an open avenue for investigation [4].

(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 over 90% accuracy solely from TLS record bursts [28]. Apthorpe demonstrated similar leakage for Android IM apps over Wi-Fi [1]. Recent machine-learning classifiers identify not only event types but also multimedia uploads and voice-over-IP handshakes in QUIC traffic [22]. Countermeasures such as adaptive padding have been proposed, which raise bandwidth by 8–14% while mitigating most classifiers [36].

(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 [41]. Kwon showed that Telegram's "People Nearby" feature enables trilateration of user location with meter-level accuracy [21]. Signal introduced private-set-intersection with SGX enclaves to curb this vector, yet follow-up work revealed side-channels via cache timing [27].

(iv) Metadata Policies and Compliance Audits

A complementary line of research 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 [14]. Bock dissected Signal's sealed-sender protocol, confirming that only sender and receiver metadata persist in a 24-hour service log but flagging the exposure of push-token identifiers to Apple/-Google [5]. Recent GDPR-oriented studies show that less than 40% of messaging apps provide complete data-export facilities despite legal requirements [19].

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 Messenger, Telegram, and Signal.

2.5 Research Gap

As the review demonstrates, prior work approaches Android behaviour analysis from one of three principal angles, each with inherent trade-offs.

Static inspection (e.g., code-review and manifest checks) is efficient and scalable but fundamentally struggles with behaviours that surface only at run time, such as those involving dynamic code loading or heavy obfuscation [24, 18].

User-space monitors—such as taint-tracking or framework hooks—offer richer semantic context. Yet, they typically require a modified or rooted runtime [69, 16], can be detected and evaded by well-crafted apps [20], and must often place implicit trust in the behavior of native code libraries.

Finally, *kernel-level tracing* delivers a low-overhead and tamper-resistant vantage point, but the raw syscalls and Binder op-codes it yields expose little direct semantic intent. This creates a considerable "semantic gap" between low-level system events and high-level application logic, a challenge actively being researched [33].

Taken together, these observations point to a clear methodological gap: there is a need for techniques that combine the high-fidelity visibility of kernel-level tracing with the contextual insights of static analysis. This gap is particularly pronounced in the domain of messaging applications. While numerous studies have analyzed these apps from a digital forensics or network-level metadata perspective [29], the application of system-wide runtime monitoring remains limited.

This thesis addresses this twofold gap by proposing a framework that not only integrates static and kernel-level dynamic analysis but also applies it systematically to profile the privacy-relevant behaviors of popular messaging apps. Designed for practical use, our system operates on stock consumer devices and, while it requires root privileges for kernel instrumentation, it avoids the need for custom firmware or kernel recompilation. It achieves its accessibility and minimizes complex configuration through a fully automated analysis pipeline and an interactive web-based dashboard for intuitive visualization of the collected behavioral evidence.

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.

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 [71]. 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 is composed of several interconnected layers, typically including the Application Layer, the Java API Framework (or Application Framework), the Android Runtime (ART) and Native Libraries, the Hardware Abstraction Layer (HAL), and the Linux Kernel itself [81].

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

Beneath the framework lies the Android Runtime (ART), which executes application bytecode and optimizes it using ahead-of-time (AOT), just-in-time (JIT), or

interpretation modes [72]. 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 [73]. 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 [80]. Android extends the kernel with additional features including the Binder IPC driver, ashmem (anonymous shared memory), and wakelocks to manage power usage [18]. This kernel foundation ensures that resource access is isolated and controlled across all system layers.

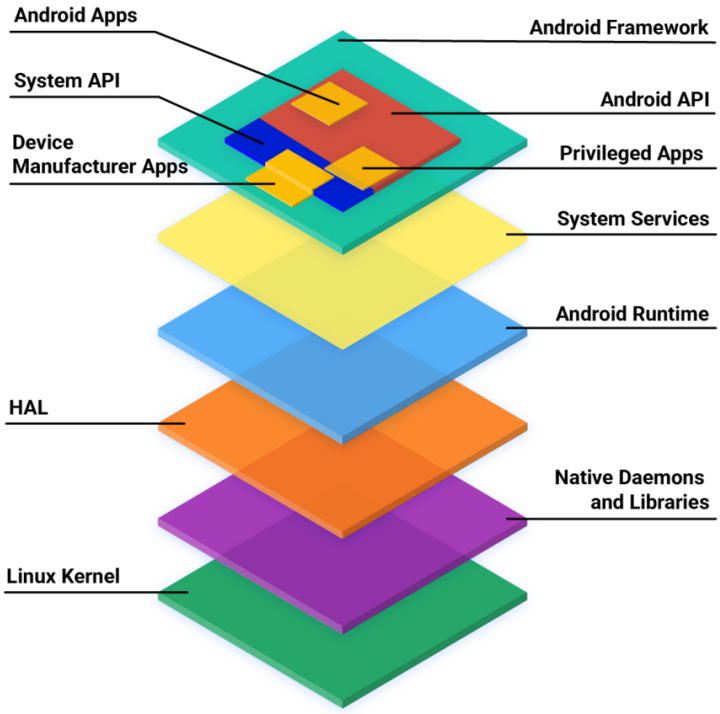

Figure 3.1: Updated Diagram of Android Software Stack (source: Android Developers Guide [71]).

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 [71].

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.

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 [72]. 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 [74]. These native operations often bypass standard framework controls, introducing low-level file, network, or cryptographic actions. This is particularly relevant for behavioral profiling, as native code may perform sensitive operations that differ from those visible at the Java level.

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

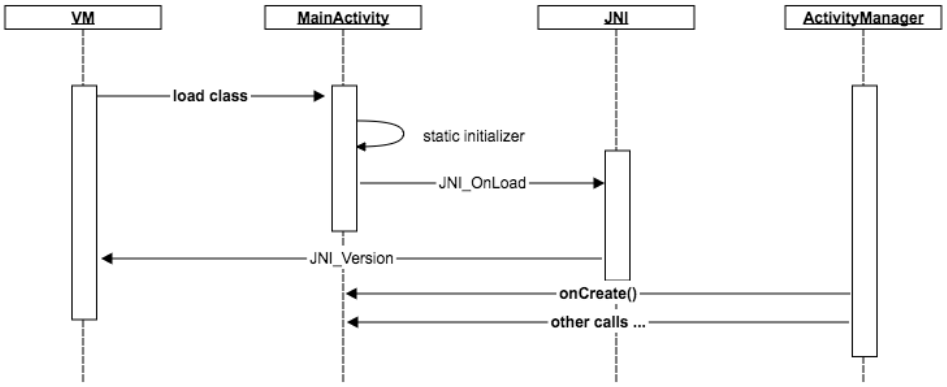

Figure 3.2: Sequence diagram showing the JNI initialization flow in an Android application.

3.1.4 Linux Kernel Fundamentals and System Calls in Android

The Android operating system relies on the Linux kernel as its foundational layer for managing hardware resources, abstracting device drivers, and ensuring secure process isolation [80]. All interactions between user applications and hardware components are mediated through the kernel's system call interface, making it a critical observation point for behavioral profiling.

Each time an application requests a low-level operation—such as reading a file, accessing a sensor, or opening a network socket—it invokes one or more system calls, which switch execution from user space to kernel space. In Android, these calls are typically issued via the Bionic libc, which serves as the system's C standard library, or directly through Java Native Interface (JNI) bindings when applications rely on native code for performance-critical tasks. Capturing these calls reveals how an application actually uses system resources, regardless of its declared permissions or documented features.

Android's kernel extends the standard Linux design with additional components such as the Binder IPC driver for efficient inter-process communication, ashmem for shared memory management, and wakelocks for fine-grained power control [71]. These features generate distinctive kernel-level events that tracing tools can capture to uncover hidden or background activities in messaging apps.

For this purpose, tracing frameworks such as `ftrace`, `kprobes`, and eBPF are employed (see Section 2.3.1 for an overview of these infrastructures). Instrumenting kernel functions such as `ksys_open` or `__sys_sendmsg` enables the collection of detailed

logs on file operations, network transmissions, and process scheduling, which form the basis for constructing robust behavioral profiles and detecting suspicious usage patterns.

3.1.5 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 [75].

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.

3.2 Messaging Apps: Characteristics and Privacy Implications

3.2.1 Functional 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 [57, 60, 59]. 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 [71].

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.

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 [76].

Resource access is another key concern. Most messaging apps request access to sensitive resources such as contacts (`READ_CONTACTS`), device 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 [16].

Additionally, messaging apps rely on push notification services such as Firebase Cloud Messaging (FCM), which has superseded 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 [39].

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()`) [29].

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.

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 [55]. 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 [30]. 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()` [39].

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.

Chapter 4

Methodology and System Design

This thesis forms part of a broader research initiative focused on the security and behavioral analysis of Android applications using combined static inspection and custom kernel-level tracing. While the present work concentrates on profiling the runtime behavior of messaging apps for privacy analysis—by extending SliceDroid and correlating static and dynamic findings—related efforts by the research group explore aspects such as automated anomaly detection, enhanced portability of tracing scripts across devices, and offset-independent instrumentation techniques. These complementary topics fall outside the scope of this thesis but inform its design and future extensions.

4.1 Research Design

The research design adopts a hybrid static–dynamic methodology to systematically uncover and characterise behavioural and security aspects of contemporary Android messaging applications without intrusive instrumentation.

4.1.1 Target Application Selection

A key design decision was the careful selection of three representative Android messaging applications: *Signal*, *Messenger* (Meta), and *Telegram*. This selection was not arbitrary but strategically motivated by the need to balance contrasting privacy models, popularity, feature diversity, and technical feasibility of analysis.

Signal was chosen as a privacy benchmark. It implements the widely respected open-source Signal Protocol, providing robust end-to-end encryption (E2EE) by design and exposing minimal metadata to intermediaries or servers. Its architecture prioritises decentralisation and ephemeral message storage, making it an ideal candidate to assess how a privacy-first approach is reflected both statically (in code) and

dynamically (in runtime traces).

Messenger, developed by Meta, was selected as a mainstream, feature-rich counterpart. It dominates global markets with billions of active users and offers a vast array of integrated features—messaging, voice/video calls, payments, and third-party integrations. This complexity translates to richer permission usage, diverse IPC pathways, and potential privacy trade-offs, serving as a high-volume, real-world stress case for the hybrid analysis pipeline.

Telegram complements the corpus by introducing a hybrid architecture: it offers both cloud-based messaging (storing messages on remote servers by default) and opt-in secret chats that utilise local device storage and ephemeral encryption keys. Additionally, Telegram supports public channels and bots, introducing unique server-client interaction patterns. Its popularity among privacy-conscious users yet fundamentally different technical approach compared to Signal enriches comparative insights.

Selecting these three applications thus ensures coverage across a spectrum: - From strict E2EE and minimum metadata (Signal), - To mainstream commercial solutions with extended functionality (Messenger), - To hybrid models blending cloud storage with privacy-enhanced modes (Telegram).

This deliberate triangulation ensures that the findings are generalisable and illustrate how different privacy philosophies and design trade-offs manifest in practical, widely deployed systems.

4.1.2 Static Analysis Methodology

Static analysis was employed as the initial phase of the hybrid approach due to its capacity to rapidly detect privacy-critical behaviors directly within application binaries without requiring execution. By systematically inspecting code structures, manifest configurations, and declared permissions, potential leakage vectors, risky API calls, and suspicious data flows can be flagged early, effectively narrowing the scope and complexity of the subsequent dynamic investigation.

For this purpose, two well-established tools were used in combination: Androguard [63] and the Mobile Security Framework (MobSF) [67]. Their integration provided both automation-friendly scripting and comprehensive security reporting capabilities. A detailed justification of why these specific tools were selected, along with their comparative strengths, is presented in Section 4.2.1 (*Tool Selection and Rationale*).

4.1.3 Dynamic Analysis Methodology

Dynamic behavioural analysis leveraged kernel-level tracing to observe runtime interactions transparently and with high fidelity. This component extended the SliceDroid framework and was deployed on a rooted Android device with Magisk and an unlocked bootloader, granting controlled access to the kernel tracing filesystem (`tracefs`). Instrumentation leveraged `ftrace`, `kprobes`, ADB utilities, shell scripts, and Python post-processors to parse raw trace data into structured JSON suitable for behavioural profiling.

A design principle prioritised simplicity and reproducibility over advanced but more invasive technologies like eBPF. The use of `kprobes` and `ftrace` alone provided sufficient observability with minimal overhead and system impact, while remaining compatible with a wide range of device kernels.

Tracing sessions were orchestrated via bash scripts deployed on-device, dynamically registering and enabling probe points targeting sensitive kernel events relevant to IPC, resource access, and networking. Extracted traces were securely offloaded, filtered by process IDs, and cleaned with heuristics to remove noise.

Finally, behavioural summaries were visualised both statically—via Python tools like `matplotlib`—and interactively, through a custom Flask–D3.js web interface supporting timeline exploration and event-based drill-down.

Detailed implementation aspects and trace artefacts are discussed in the following 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 bytecode, 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 permissions, 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 4.1.

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

Listing 4.1: Rooting commands for device setup

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 4.2.

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

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

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

Device Node Mapping for Hardware Resources

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

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

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

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

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

Listing 4.4: Encoding device number using major and minor

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

Automation and Metadata Serialization

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

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

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

Listing 4.5: Serialized JSON metadata for sensitive resource mapping

4.3.2 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 4.1 illustrates the overall architecture and the principal data flow across its components.

At a high level, the pipeline begins with a mobile-side tracing engine, which leverages configurable instrumentation via `kprobes` and declarative tracing policies. The

engine is supported by a resource resolution layer that augments raw trace events with context such as inode paths, socket identifiers, and mapping metadata. This layer plays a critical role in enabling high-level semantic interpretation of low-level events. In particular, it facilitates the identification of accesses to sensitive device resources such as the camera or microphone, as well as interactions with privacy-critical application components, including contact lists, messaging services, and storage providers.

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

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

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

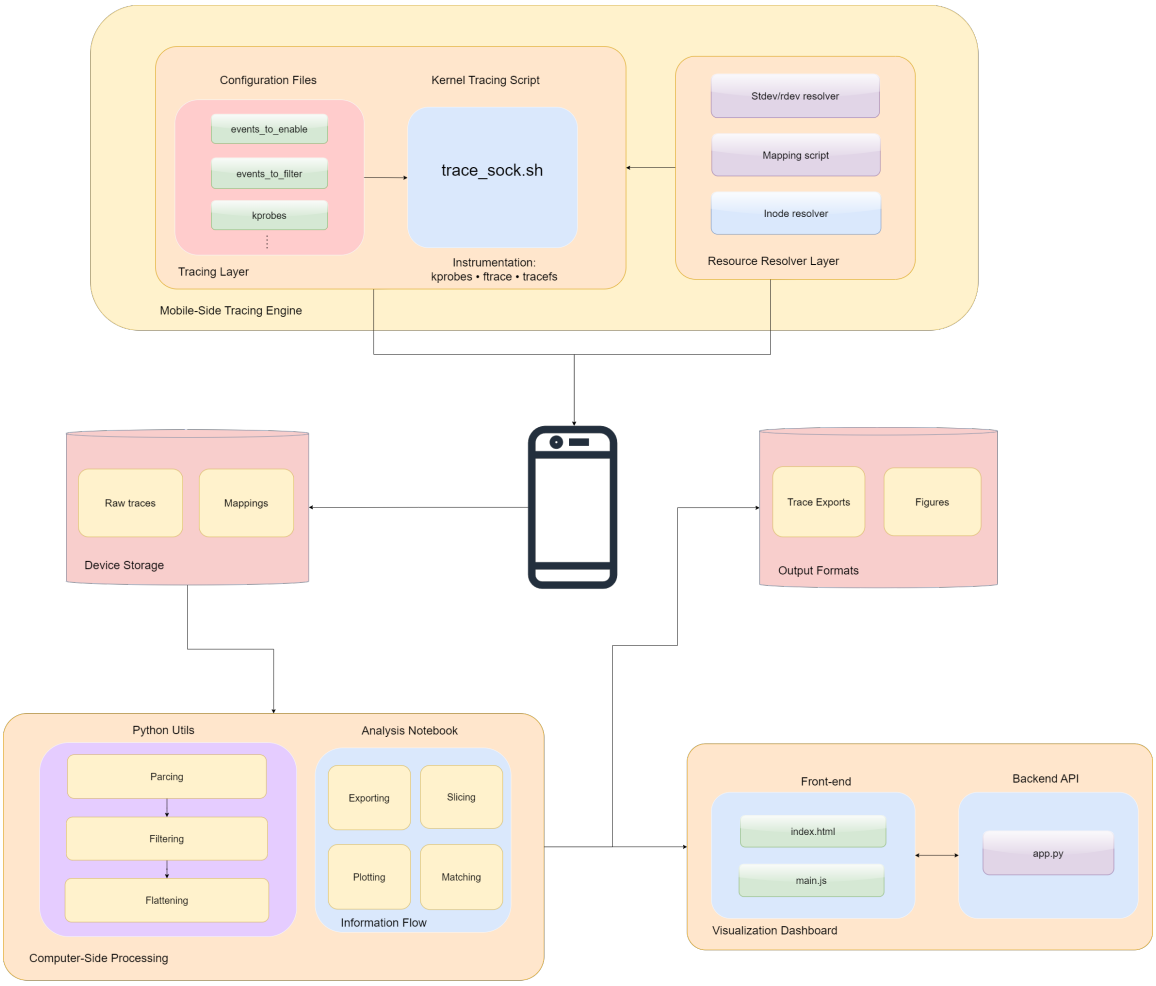

Figure 4.1: High-level modular architecture and data-flow pipeline of SliceDroid.

Codebase Structure

The codebase is organised into clearly defined top-level components to support trace collection, processing, and interactive exploration. The main elements are outlined below:

`scripts/` Contains tracing and resource-mapping scripts. It includes configuration files for enabled events, kprobes, and process filters, helper scripts for resolving device and inode identifiers, as well as the App Mapper script responsible for generating the JSON mapping of Android package names to commercial application names.

`run_slicedroid.py` Entry point for running the customised SliceDroid tracer with the specified settings.

`data/` Serves as the central storage for raw traces, identifier mappings, and processed

outputs. It includes the `Exports/` subfolder, which holds cleaned trace exports and generated figures.

`webapp/` Hosts the web-based visualisation tool, with a Flask backend and a Bootstrap/D3.js frontend for displaying behavioural timelines and analytics.

Instrumentation Layer

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 SliceDroid project at AUEB, and forms the foundational backbone of this thesis. This base layer leverages dynamic kernel probes (`kprobes`) and static `tracepoints` exposed via the `ftrace` interface, allowing efficient interception of system call and function-level activity in the kernel.

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

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

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

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

Listing 4.6: Example kprobe for vfs_write in tracefs syntax

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

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

Resource Resolver Layer

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

1. **Special-device nodes**: 32-bit packed `dev_t` values (`k_dev`) extracted from `vfs_read`/`vfs_write`/`ioctl` probes.

2. **Regular files**: `(st_dev, inode)` tuples identifying SQLite databases that hold privacy-critical data (`contacts2.db`, `mmssms.db`, *etc.*).

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

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

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

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

In addition to these streams, a key contribution of this thesis within this layer is the development of a dedicated *App Mapper Module*. The mapper resolves Android package names (e.g., `com.facebook.orca`) into recognizable commercial application names (e.g., `Messenger`). It connects to the target device via ADB to enumerate user-installed applications, parses APK metadata through static analysis using the `androguard` library, and stores the extracted results in a JSON mapping file. This mapping is subsequently utilized by downstream tracing modules and the visualization dashboard, significantly enhancing the interpretability of kernel-level events and collected trace data (see Appendix A for implementation details).

Parsing and Pre-Processing

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 SliceDroid 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 SliceDroid (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 $V =$ PIDs and $E = \{\text{read, write, ioctl, socket, binder}\}$. Graphs are merged across consecutive windows via identical transaction IDs (Binder) and socket tuples, producing a whole-session trace of:

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

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

4.3.3 Data Visualization and User Interface Design

The SliceDroid platform has been significantly extended into a fully-featured web application with an integrated dashboard, while maintaining its thoughtfully designed, step-by-step interface that guides analysts from initial trace file upload through to detailed security insights and forensics.

Figure 4.2 shows the home screen, which acts as the starting point for each analysis session. Users can upload a new raw `.trace` file by dragging it onto the upload area or selecting it manually. For convenience, the system also supports preloading the

most recently used trace file, indicated as `trace.trace`, so that repeated uploads are not necessary during iterative investigations. In parallel, the user can choose which specific application's behaviour to examine from a curated list of popular apps (e.g., Messenger, Signal, Telegram, Termux), ensuring that only relevant events are shown in the analysis views. Additional process ID and device filters refine the scope even further before launching the analysis.

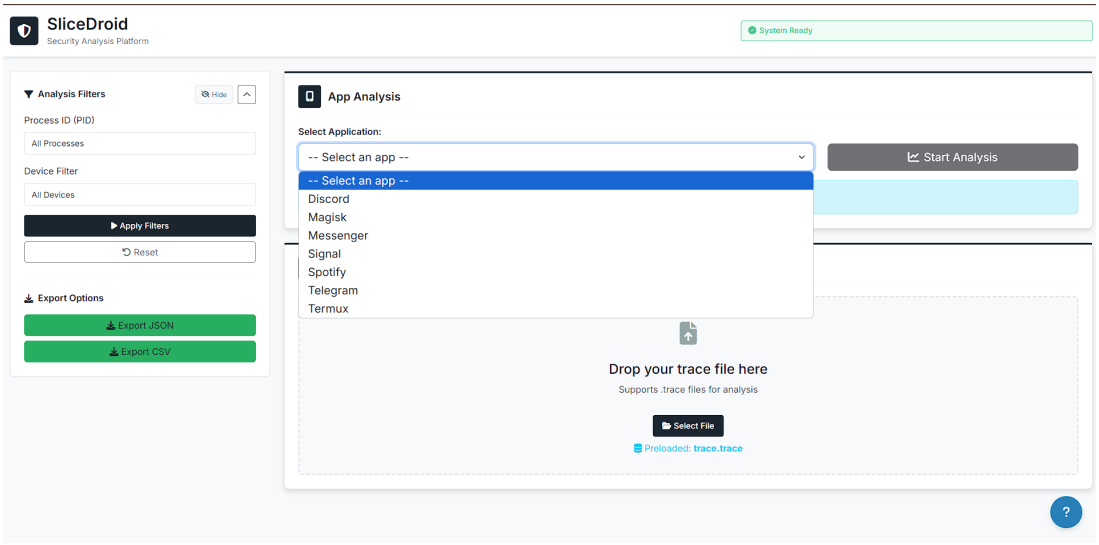

Figure 4.2: Home screen with options for uploading a new trace file, preloading the last trace, and selecting a target application for focused security analysis.

After selecting the trace and target application, the analyst transitions to the interactive event timeline (Figure 4.3). This timeline provides a clear, category-separated view of syscall activity over time, grouped into logical classes such as `read`, `write`, `ioctl`, Binder events, and network operations. The timeline supports zooming, panning, and detailed inspection via hover tooltips, allowing investigators to pinpoint unusual spikes, suspicious sequences, or gaps in expected process behaviour.

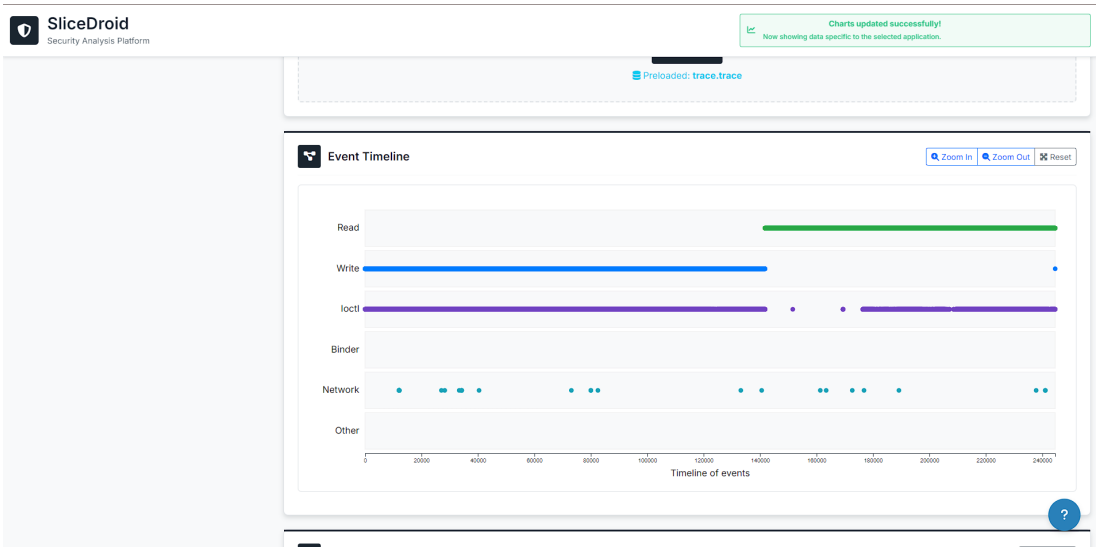

Figure 4.3: Interactive event timeline showing syscall activity split into categories with intuitive zoom and hover inspection tools.

Complementary to the timeline is the Device and Event Statistics panel (Figure 4.4). On the left, a pie chart and table break down which hardware devices were accessed during the trace period, highlighting the top devices by event count. On the right, the system shows the proportion of syscall event types (e.g., `ioctl_probe`, `read_probe`), their absolute counts, and percentage shares. This dual view helps analysts understand which resources and syscall classes dominate the workload, which is often critical for detecting outlier device usage or unexpected access patterns.

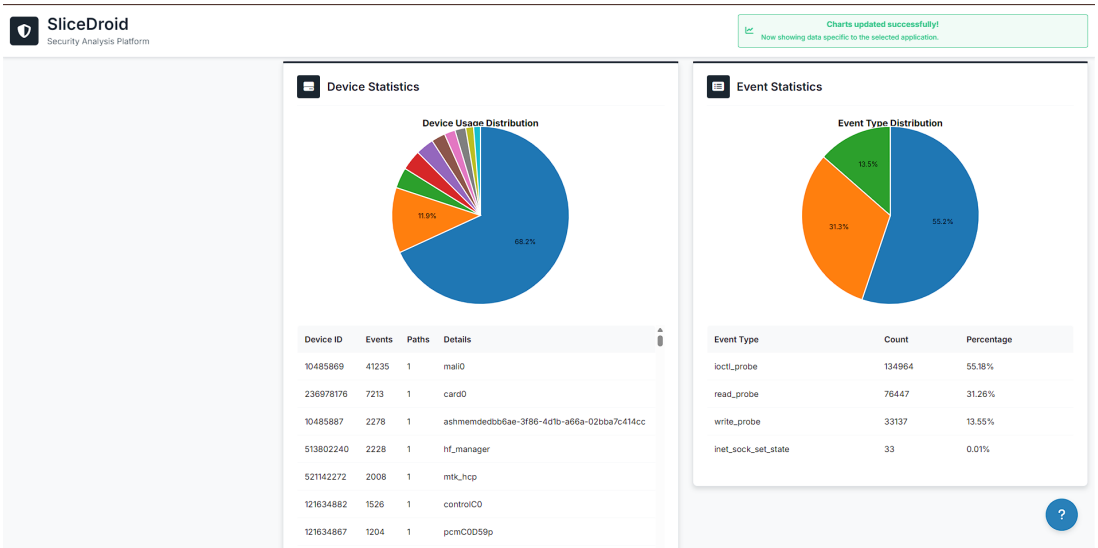

Figure 4.4: Device and event statistics: left pane shows device usage distribution and top devices; right pane displays breakdown of syscall event types.

Figure 4.5 presents the Advanced Analytics module. This section displays high-

level metrics including total event count, unique processes and devices, trace duration, and calculated event throughput in events per second. Below these KPIs, the Behaviour Timeline plots activity windows for categories like Camera, TCP, Bluetooth, SMS, and other sensitive subsystems. This windowed view helps correlate spikes in device behaviour with specific time windows, enabling fine-grained behavioural profiling.

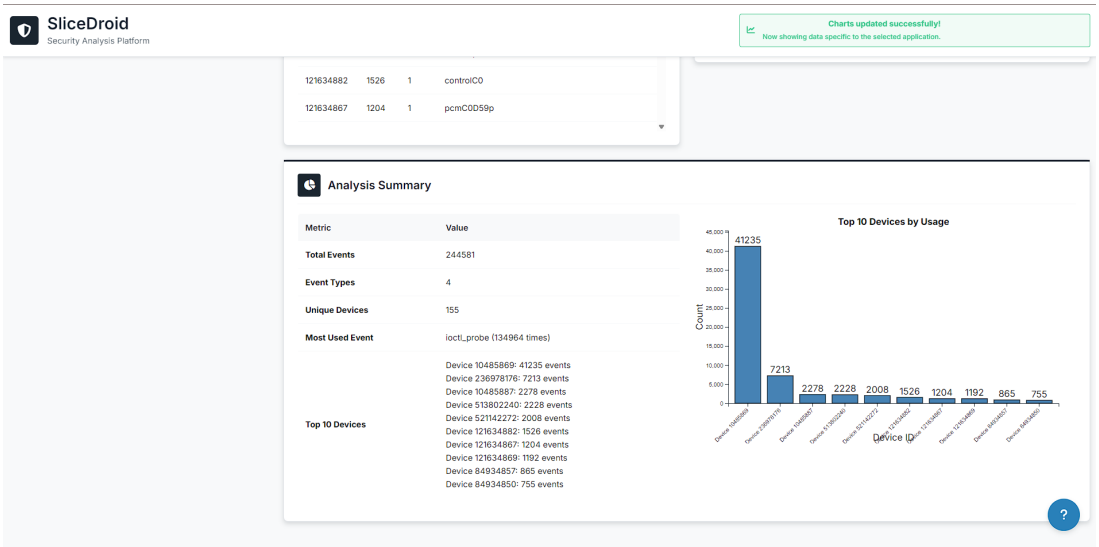

Figure 4.5: Advanced Analytics: high-level metrics and the Behaviour Timeline, offering windowed insight into sensitive categories such as network and peripheral usage.

For focused inspection of network behaviour, the Network Analysis module (Figure 4.6) visualises the distribution of TCP and UDP events, socket operations, and the protocols in use. A graph-based Communication Flow diagram depicts the relationships between processes and external endpoints. Such visual context allows the investigator to detect unexpected data transfers or suspicious inter-process communication pathways.

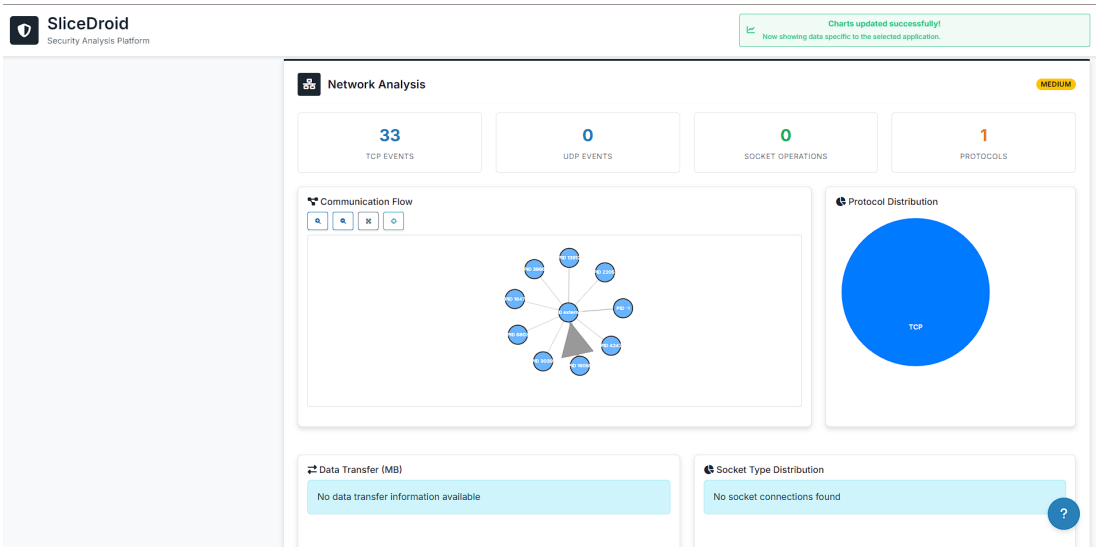

Figure 4.6: Network Analysis: Communication Flow diagram, protocol distribution pie chart, and connection statistics for tracking network behaviour and potential exfiltration vectors.

Process genealogy is equally crucial. Figure 4.7 illustrates how the system reconstructs the parent-child process tree, showing process activity summaries including PIDs, event counts, and suspicious patterns (highlighted below the tree). Here, anomalies such as excessive file accesses or unexpected forks can be quickly identified and traced back to their origin.

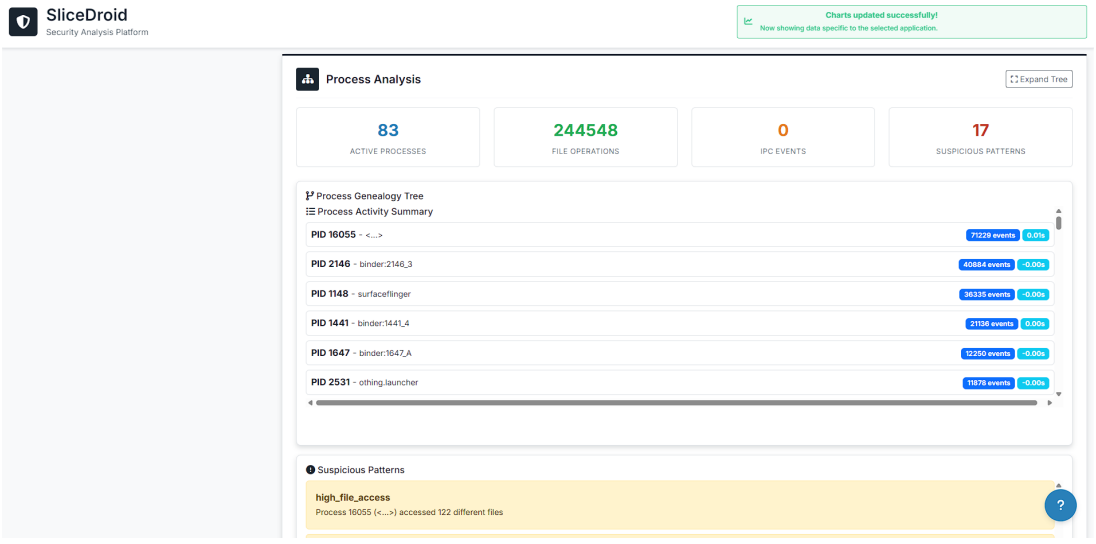

Figure 4.7: Process Analysis: dynamic genealogy tree with activity summary and detected suspicious patterns, such as high file access counts.

Finally, Figure 4.8 shows the dashboard's integrated security summary. This view aggregates risk scores, highlights privilege escalation attempts, debugging traces, or

policy violations, and provides contextual recommendations for remediation. By centralising these indicators, analysts can prioritise threats and document findings more effectively.

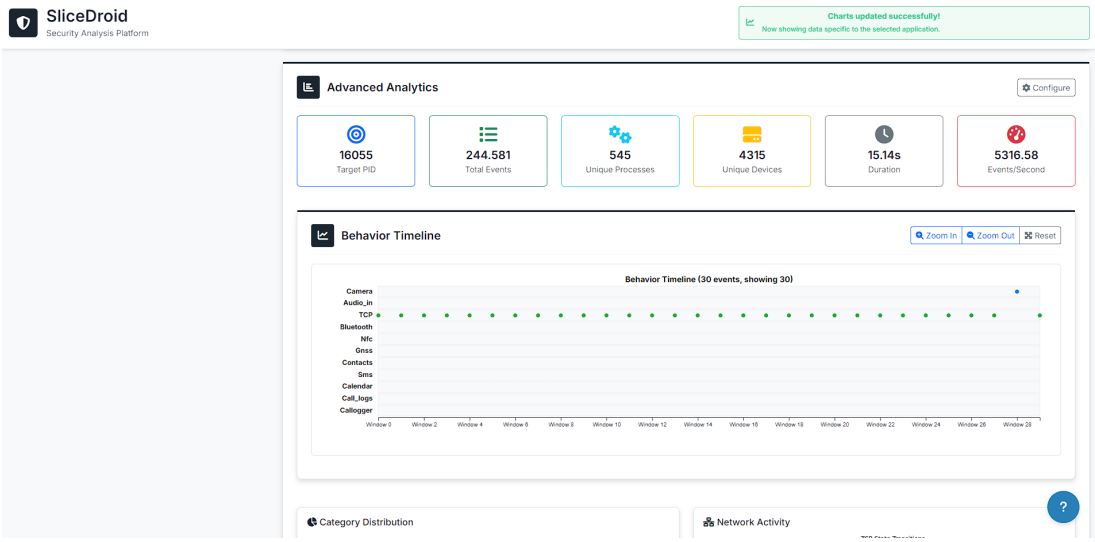

Figure 4.8: Security Overview: centralised display of detected risks, privilege escalation attempts, and recommended actions for the analyst.

In summary, the combined views — including the timeline, device and event statistics, advanced analytics, network and process modules, and security overlays — converge in a comprehensive summary dashboard (Figure 4.9). This final snapshot condenses category distribution, TCP state transitions, top device usage, and most active processes into a single, intuitive screen, offering a clear, actionable overview for immediate security insight and reporting.

Figure 4.9: Comprehensive Summary Dashboard: unified visual of event categories, TCP states, device usage, and active process distribution for rapid situational awareness.

Technically, all dynamic charts are rendered using `D3.js`, and the responsive UI is built with `Bootstrap`. A collapsible sidebar, integrated help tooltips, and live system health checks ensure usability and reliability throughout long-duration trace analysis sessions.

Pipeline Summary

To close this architecture chapter, Figure 4.10 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.

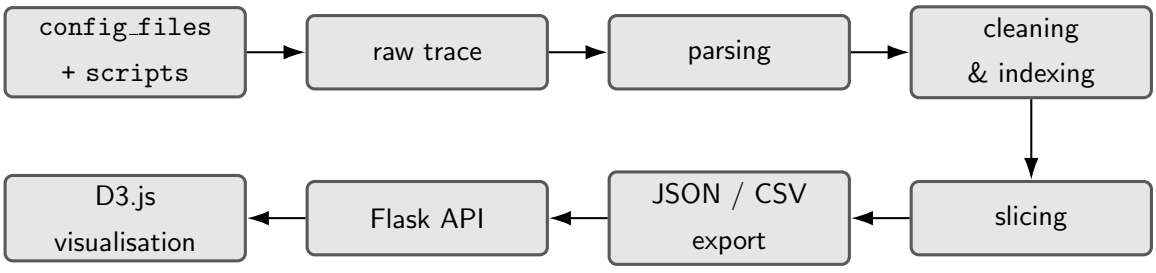

Figure 4.10: End-to-end data flow from probe configuration to interactive analytics.

4.3.4 Workload Design and Usage Scenarios

To capture meaningful behavioural variations and to systematically compare how each application responds under distinct runtime conditions, dynamic kernel-level tracing was organised into four carefully defined usage scenarios per target app. This multi-scenario design enables direct cross-scenario comparison, revealing differences in resource usage, inter-process communication (IPC), and network interactions when permissions or user activity levels vary.

i. **Full Permissions Scenario:** The application was run with all permissions granted, allowing observation of its maximum functional scope. In this mode, a consistent sequence of actions was performed: sending two text messages, recording a short voice message (approximately 5 seconds long), and capturing a photo using the in-app camera. This interaction pattern ensures coverage of typical real-world usage and tests multiple privacy-sensitive features simultaneously. The full permissions scenario serves as a baseline to compare against restricted and passive states, helping isolate which activities depend on specific permissions.

ii. **Restricted Permissions Scenario:** Selected sensitive permissions (e.g., location, microphone, or contacts) were deliberately disabled through the Android settings. This condition was designed to examine how the application adapts when permission-dependent features are blocked—whether it attempts repeated access, triggers fallback code, or suppresses functionality altogether. Comparing this scenario to the full permissions case highlights how gracefully or aggressively each app handles denied permissions.

iii. **Background Passive Scenario:** The app was left running in the background with normal permissions but without user interaction. This scenario was included to detect silent background behaviours such as periodic syncs, push notification handling, or unexpected network calls that occur independently of user input. Analysing this passive state alongside active usage reveals latent behaviours that could impact privacy or battery consumption.

iv. **Active Foreground Scenario:** The app was actively used for normal tasks such as messaging, media sharing, and voice or video calls where applicable. This scenario represents the peak operational state with frequent user actions, providing a reference for typical real-world usage patterns. Comparing it with the passive and restricted runs helps pinpoint which kernel-level events are triggered exclusively by user-initiated activity versus background processes.

Each scenario execution lasted approximately one minute to capture sufficient behavioural samples and was repeated multiple times to enhance reliability and minimise outlier effects. The runs were systematically scheduled in a consistent sequence—starting from passive, followed by restricted, then full permissions, and concluding with active usage—to ensure stable device conditions and to avoid cross-contamination of residual states.

By applying this structured workload design, the resulting traces enable detailed side-by-side comparisons. This makes it possible to clearly identify behavioural deviations across permission settings and activity states, providing deeper insights into each application's runtime privacy implications and resource footprint.

Chapter 5

Results

5.1 Static Analysis Results

This chapter presents a detailed *static* security appraisal of three widely used instant messaging clients—*Signal*, *Telegram*, and *Meta Messenger*—based on production-signed APK packages acquired directly from physical Android devices using the ADB. Specifically, each APK was retrieved via the following commands:

First, all installed packages on the device were enumerated with:

```
adb shell pm list packages
```

Next, the full installation path of a selected package was queried:

```
adb shell pm path <package_name>
```

Finally, the APK was pulled to the local workstation:

```
adb pull /data/app/<apk_path> ./
```

The subsequent static evaluation utilises tools including MobSF, Androguard, and APKID to thoroughly dissect manifest configurations, bytecode payloads, native binaries, and embedded resources.

5.1.1 Scope and Dataset Footprint

The static security review was systematically conducted in five incremental stages to ensure breadth and depth of inspection.

Manifest Analysis scrutinised high-risk configuration flags—most notably the `allowBackup` attribute, the exposure of components via exported activities and services, and the full set of declared Android permissions.

Bytecode Inspection involved decompiling and scanning the compiled `.dex` code, pinpointing insecure coding constructs, potential logic flaws, and signatures of known vulnerabilities.

Native Library Hardening assessed the integrity of any packaged native binaries, verifying the presence of critical exploit mitigations such as Non-Executable Stack (NX), Position-Independent Executable (PIE), Stack Canaries, RELRO (Relocation Read-Only), and FORTIFY source hardening.

Secret Scanning leveraged entropy-based heuristics to detect embedded sensitive data, such as API tokens, secret keys, and credentials inadvertently hard-coded in resources or source strings.

Finally, **Anti-Analysis and Obfuscation Review** catalogued defensive mechanisms against dynamic or static reverse engineering, including anti-debugging checks, emulator detection routines, and custom obfuscation artefacts.

The artifacts from this multi-phase assessment were compiled as succinct CSV exports (each under 100 kB) and comprehensive PDF reports (typically between 100 and 200 kB).

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

Table 5.1: Approximate sizes of static analysis output files

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 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 66 MB, markedly smaller than Signal's 86 MB. 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.

Table 5.2: 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 5.3: 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 hun-

dred 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 5.4: 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 Table 5.5, 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.

Table 5.5: Manifest Issues by Severity (MobSF)

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

Turning to the individual flags in Table 5.6, all three applications keep `allowBackup=true`, meaning a device in developer mode can exfiltrate private app data unless 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 5.6: 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.

Table 5.7: 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`	—	✓	✓

Bytecode-level exposure. Static inspection of the `.dex` payloads (Table 5.8) confirms a common baseline of cryptographic missteps—most notably the continued use of AES in CBC mode without accompanying integrity checks. Both **Messenger** and **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.

Table 5.8: Comparative Matrix of Code–Level Vulnerabilities

Issue	**Messenger**	**Telegram**	**Signal**
High-severity findings	4	5	3
Remote-debuggable `WebView`	✓	✓	✗
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

Third-party telemetry *vs.* privileged scope. Table 5.9 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.

Risk, however, is not dictated by trackers alone. Table 5.10 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 5.9: Declared third-party tracker SDKs

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

Table 5.10: Overlap with 25 malware-associated permissions

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

Firebase 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 Firebase endpoints appear in the binary. Static string inspection, on the other hand, reveals hard-coded Firebase 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 5.11: 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

The dynamic analysis was systematically conducted to capture runtime behaviour under the exact usage scenarios previously defined and executed in Section 4.3.4 *Workload Design and Usage Scenarios*, using the selected messaging applications. Each application was exercised in a controlled environment for approximately 30 ± 1 seconds per run, under four distinct operational modes: running in the background with full operational permissions (`full_background`), actively used in the foreground with full permissions (`full_foreground`), background execution with minimal resource activity (`none_background`), and foreground usage with reduced permissions (`none_foreground`). This workload configuration ensures behavioural consistency

and repeatability across all test runs. The resulting dataset comprises twelve trace files, each clearly named to encode the specific application, scenario, and recording duration, providing full traceability and alignment with the workload structure. Observed file sizes demonstrate an expected trend: foreground traces under full permissions exhibit significantly larger footprints due to richer event flows from user interactions and multimedia activities, while background and constrained scenarios yield notably smaller trace volumes, reflecting lower runtime resource engagement.For example, the trace for full foreground usage is up to 117 KB, compared to 16 KB for a none background case. This structured dataset reflects the defined workload scenarios and provides a reliable basis for comparing application behaviour, resource usage, and runtime characteristics under consistent operating conditions. Notably, Signal consistently generates a smaller trace size than the other applications in the foreground scenarios, which may indicate more minimalistic runtime behaviour and stricter resource handling, in line with its lean and security-focused design previously highlighted in the static analysis. This aspect will be further examined in the following sections, where dynamic and static findings are jointly discussed.

Table 5.12: Grouped Overview of Trace Durations and File Sizes

Scenario Type	Context	Application	Duration (sec)	Size (KB)
Full	Background	Messenger	30.01	15.286
		Signal	29.57	15.155
		Telegram	29.83	15.670
	Foreground	Messenger	30.17	117.109
		Signal	29.02	87.952
		Telegram	29.52	115.274
None	Background	Messenger	29.00	16.319
		Signal	29.46	16.376
		Telegram	30.39	16.547
	Foreground	Messenger	30.35	72.231
		Signal	30.46	66.798
		Telegram	30.19	75.101

5.2.2 Statistical Summaries

Messenger Foreground Runtime Behaviour

This section provides a concise descriptive snapshot of Messenger when running under typical user-driven interaction. As outlined in the *Workload Design and Usage Scenarios*, the app was actively used in the foreground for sending text messages, recording a short voice clip, and capturing an image using the in-app camera. This

representative workload was chosen as it reflects everyday usage, maximising feature invocation and generating diverse runtime events for meaningful analysis.

During this active usage interval of approximately 30 seconds, a total of 557, 770 events were recorded, yielding an average throughput of about 18, 512 events per second. This high event density highlights Messenger's intensive interaction patterns and concurrent background tasks typical of a realistic foreground session. The main runtime statistics summarised in Table 5.13 show that over a thousand processes were spawned, with a notable volume of file operations and IPC calls.

Table 5.13: Key Runtime Statistics for Messenger

Metric	**Value**
Duration	30.17 sec
Total Events	557,770
Events per Second	18,512
Unique Processes	1,184
Active Processes	218
File Operations	410,707
IPC Events	85,836
Notable Anomalies	27 distinctive patterns
Read/Write Ratio	1.73
Most Accessed File Type	.db (7,693 times)

As shown, file operations dominate with over 410, 000 I/O calls, while inter-process communication contributes another 85, 836 events. The so-called *distinctive patterns* in Table 5.13 refer to behavioural outliers, for example when an otherwise idle process suddenly performs an unusually high number of writes; these are not inherently malicious but can serve as an early hint for deeper inspection.

In terms of network behaviour and event type proportions, the `ioctl` category accounts for nearly half of all recorded calls, followed by `binder`, `read`, and `write` operations. Messenger also generates steady TCP traffic with a small number of UDP packets, transferring around 3.48 MB over TCP, alongside significant local UNIX datagram exchanges that reflect intensive intra-app service communication.

Table 5.14: Network and Event Category Breakdown for Messenger

Network Metric	**Value**
TCP Events	750
UDP Events	2
Total TCP Data	3.48 MB (0.3 MB sent, 3.16 MB received)
UNIX Datagram Traffic	2.44 MB sent, 48.2 MB received
Unique Protocols	4
Unique Connections	2
Event Category	**Proportion**
ioctl	46.6%
binder	18.5%
read	18.1%
write	10.5%
network	6.3%

Taken together, these figures confirm Messenger's reliance on high-volume local storage, inter-process orchestration, and steady external connectivity during realistic usage—offering a robust baseline for comparing runtime patterns under constrained or passive states in subsequent sections.

Messenger Background Runtime Behaviour

When Messenger was left running in the background without direct user interaction, the captured trace covered a comparable window of about 30 seconds and produced a total of 38,965 events. This results in a much lower average throughput of approximately 1,300 events per second—significantly reduced compared to the 18,512 events per second observed during active usage.

From a process perspective, only 42 processes remained active in the background compared to 218 during the foreground session, which is entirely expected given the absence of active user interaction and the reduced need for real-time service coordination. File operations continued to dominate system calls, with 38,955 I/O events logged. The read/write ratio increased to 1.50, indicating a clear preference for read operations over writes—typical of periodic local state checks or passive data syncing rather than active message composition or media uploads. Database files (`.db`) remained the most accessed type but were triggered just 2,284 times, confirming limited storage engagement.

Regarding IPC, no explicit inter-process communication events were detected during this idle phase, suggesting that Messenger's background services rely mostly on

persistent resident processes rather than dynamic component interaction. Nonetheless, 8 distinctive patterns were still flagged, representing short bursts where isolated processes performed unusual spikes of read or ioctl calls—these do not indicate malicious behaviour but highlight passive checks or background sync routines.

Network activity in the background was minimal: only 8 TCP events and 2 UDP packets were detected, transferring less than 0.01 MB in total. This aligns with lightweight push notification checks or silent polling for updates—behaviour that is typical for messaging apps that need to remain responsive while conserving bandwidth and battery.

Overall, Messenger's runtime footprint drops significantly when idle, demonstrating efficient resource reduction while maintaining enough background activity to ensure timely delivery of new messages and notifications without aggressive data usage.

Table 5.15: Key Runtime Statistics for Messenger (Background)

Metric	**Value**
Duration	30.01 sec
Total Events	38,965
Events per Second	1,298
Unique Processes	175
Active Processes	42
File Operations	38,955
IPC Events	0
Distinctive Patterns	8 instances
Read/Write Ratio	1.50
Most Accessed File Type	.db (2,284 times)
TCP Events	8
UDP Events	2
Total TCP Data	< 0.01 MB
UNIX Datagram Traffic	Not detected

Signal Foreground Runtime Behaviour

Signal demonstrates a notably lighter runtime signature compared to Messenger when operated under equivalent user-driven conditions.

Over an active usage window of about 30 seconds, Signal generated a total of 423,442 recorded events, yielding an average throughput of approximately 14,600 events per second—considerably lower than Messenger's event rate, underscoring Signal's leaner process structure and restrained resource use. Table 5.16 summarises the

core runtime metrics, showing that fewer unique processes were spawned and active at runtime.

Table 5.16: Key Runtime Statistics for Signal

Metric	**Value**
Duration	29.02 sec
Total Events	423,442
Events per Second	14,600
Unique Processes	546
Active Processes	123
File Operations	332,144
IPC Events	50,680
Distinctive Patterns	8 instances
Read/Write Ratio	1.03
Most Accessed File Type	.db (3,248 times)

File I/O operations account for most of the trace, supported by moderate IPC exchanges. The few *distinctive patterns* noted indicate rare spikes in activity—such as isolated processes performing unexpected batches of operations—which are not inherently harmful but can warrant closer review during forensic analysis.

Regarding network behaviour and system call proportions, Signal recorded 263 TCP events, transferring about 1.62 MB (0.12 MB sent, 1.49 MB received), and showed no UDP traffic. Interestingly, no UNIX datagram traffic was captured during this session, which may reflect either the app's design to avoid local service sockets or a limitation of the tracing setup to fully detect certain IPC channels. This point is acknowledged as a measurement constraint and is discussed further in the limitations section. The `ioctl` category accounts for over half of all calls, with balanced shares for `binder`, `read`, and `write` operations, as summarised below.

Table 5.17: Network and Event Category Breakdown for Signal

Network Metric	**Value**
TCP Events	263
UDP Events	0
Total TCP Data	1.62 MB (0.12 MB sent, 1.49 MB received)
UNIX Datagram Traffic	Not detected
Unique Protocols	3
Unique Connections	1
Event Category	**Proportion**
ioctl	54.4%
binder	14.4%
read	13.4%
write	13.0%
network	4.8%

In summary, these results confirm that Signal's runtime footprint remains significantly lower than Messenger's, prioritising secure transmission and minimal local overhead. This behaviour aligns well with its privacy-focused design, complementing insights from static analysis.

Signal Background Runtime Behaviour

When Signal operates in the background, its runtime footprint remains minimal compared to its active usage, which is logical and expected for a security-focused application designed to suppress unnecessary background processing. Over an observation window of approximately 29.5 seconds, a total of 30,663 events were recorded, corresponding to an average throughput of about 1,039 events per second—highlighting a substantial drop in runtime intensity relative to the foreground session.

The system call distribution shows a very high read-to-write ratio of 168.09, indicating that almost all operations consist of reading pre-existing local state, verifying data integrity, or performing lightweight consistency checks without actively writing back to storage. This behaviour demonstrates how Signal prioritises passive checks (such as validating message status or verifying session keys) while strictly avoiding unnecessary file modifications, aligning with its minimal data retention and privacy-first design.

File operations remain the dominant activity, while network communication is practically absent: only 5 TCP events were observed, with no UDP or other local socket activity detected. Inter-process communication was similarly inactive during

this idle window, confirming that Signal's background logic does not rely on frequent module chaining or service calls.

Compared to Messenger's background trace, Signal's passive footprint is significantly lower in all categories—fewer active processes, drastically fewer network requests, and a clear emphasis on read-only local checks. These observations, summarised in Table 5.18, reinforce Signal's commitment to minimising persistent background presence and reducing potential metadata exposure.

Table 5.18: Key Runtime Statistics for Signal (Background)

Metric	**Value**
Duration	29.5 sec
Total Events	30,663
Events per Second	1,039
Unique Processes	134
Active Processes	36
File Operations	30,658
IPC Events	0
Distinctive Patterns	8 instances
Read/Write Ratio	168.09
Most Accessed File Type	.db (2,264 times)
TCP Events	5
UDP Events	0
Total TCP Data	0.02 MB (0.00 MB sent, 0.02 MB received)
UNIX Datagram Traffic	Not detected

Telegram Foreground Runtime Behaviour

Telegram shows a noticeably heavier runtime profile than Signal and a closer footprint to Messenger when used under typical user-driven conditions. The test session included sending messages, recording a short voice note, and capturing an image to ensure realistic, feature-rich interaction.

Across an active usage window of approximately 30 seconds, Telegram generated a total of 549,824 recorded events, resulting in an average throughput of about 18,625 events per second. This high rate, summarised in Table 5.19, indicates an application architecture optimised for rapid background operations and parallel handling of media-rich content.

Table 5.19: Key Runtime Statistics for Telegram

Metric	**Value**
Duration	29.52 sec
Total Events	549,824
Events per Second	18,625
Unique Processes	789
Active Processes	238
File Operations	460,164
IPC Events	47,629
Distinctive Patterns	28 instances
Read/Write Ratio	1.39
Most Accessed File Type	.db (5,302 times)

File I/O is again dominant, with a slight bias towards read operations. IPC activity was substantial at 47,629 events, and 28 *distinctive patterns* were flagged—meaning isolated processes showed sudden spikes in operations that stand out from the main activity flow, but do not imply malicious behaviour on their own.

Network-wise, Telegram initiated 462 TCP events and 3 UDP packets, transferring a total of 574.36 MB over TCP, of which 0.26 MB was sent and 574.10 MB was received. However, the unusually high inbound network traffic—574.10 MB received during a short interaction window—most likely results from a framework-related logging artefact or misattribution bug, rather than genuine application behaviour. As such, it will not be further analysed in the subsequent evaluation, but is noted here for completeness. The absence of UNIX datagram traffic may again reflect either the app's internal design or a limitation in the capture configuration. This large inbound traffic likely results from automatic media downloads, large attachment fetching, or possibly a background update triggered during the test window—an aspect noted for discussion in the limitations section. Event type proportions show that `ioctl` calls remain dominant (57.3%), with read, write, and binder operations distributed evenly below that threshold.

Table 5.20: Network and Event Category Breakdown for Telegram

Network Metric	**Value**
TCP Events	462
UDP Events	3
Total TCP Data	574.36 MB (0.26 MB sent, 574.10 MB received)*
UNIX Datagram Traffic	Not detected
Unique Protocols	4
Unique Connections	5
Event Category	**Proportion**
ioctl	57.3%
read	16.8%
write	12.1%
binder	10.1%
network	3.7%

* This unusually high inbound value is likely a logging artefact and is excluded from further analysis.

Taken together, these results indicate that Telegram exhibits a consistently high I/O and inter-process communication (IPC) footprint, likely reflecting its architecture's emphasis on media-rich interactions and continuous background synchronization.

Telegram Background Runtime Behaviour

When left running passively in the background, Telegram retains a moderately active runtime footprint, as expected. Over an observation window of approximately 30 seconds, the background session produced 39,340 total events, translating to about 1,318 events per second—representing roughly a 93% reduction compared to its active throughput.

File operations dominate the trace, with 39,336 I/O events and a balanced read-/write ratio of 1.45. Local database files (`.db`) remained the primary storage target, accessed 2,043 times, which reflects background cache refreshes or silent content syncing.

No inter-process communication was detected, contrasting with Telegram's interactive use where IPC is more pronounced. Six distinctive patterns were flagged—brief spikes or repetitive calls that deviate slightly from the baseline but do not indicate suspicious behaviour.

Network usage stayed minimal: only 4 TCP events were logged and no UDP

activity, with negligible data transferred (< 0.02 MB), suggesting occasional keep-alive connections or push checks.

Overall, as summarised in Table 5.21, Telegram's background activity is more persistent than Signal's near-silent idle mode but broadly similar to Messenger's residual maintenance footprint.

Table 5.21: Key Runtime Statistics for Telegram (Background)

Metric	**Value**
Duration	29.83 sec
Total Events	39,340
Events per Second	1,318
Unique Processes	194
Active Processes	42
File Operations	39,336
IPC Events	0
Distinctive Patterns	6
Read/Write Ratio	1.45
Most Accessed File Type	.db (2,043 times)
TCP Events	4
UDP Events	0
Total TCP Data	≈ 0.02 MB

Cross-Application Comparison

When comparing the three applications under identical high-activity foreground scenarios, clear behavioural differences emerge. Messenger exhibits the highest overall file operations and IPC activity, reflecting its more complex interaction flows and heavier reliance on local data handling during messaging and multimedia tasks. Signal, in contrast, maintains the smallest runtime footprint in both I/O and network usage, aligning with its minimalistic and privacy-oriented architecture. Telegram also demonstrates elevated file and IPC activity, consistent with its emphasis on background synchronization and preloading mechanisms.

While the recorded network throughput for Telegram exceeded 570 MB in a single session, this value is likely the result of a framework-level artefact and is therefore excluded from comparative conclusions. Nonetheless, previous analyses suggest that Telegram tends to prefetch media content and synchronise message history more aggressively than its counterparts, a pattern still observable through non-network signals.

Category distributions further confirm that while all three applications are dominated by `ioctl` and file operations, Telegram and Messenger maintain a more balanced proportion of `binder` and network calls, highlighting richer IPC layers and more frequent background exchanges than Signal.

Overall, this cross-app comparison underscores distinct architectural choices: Messenger favours constant local activity for performance and feature richness; Signal minimises runtime footprints to maximise privacy guarantees; and Telegram invests heavily in preloading and synchronization for a seamless content experience. These findings align consistently across both foreground and background measurements—excluding anomalous network values—and support the descriptive and statistical conclusions presented in the previous sections.

Table 5.22: Comparison of Key Runtime Metrics Across Applications (Full Foreground Scenario)

Metric	**Messenger**	**Signal**	**Telegram**
Duration (sec)	30.13	29.02	29.52
Total Events	557,770	423,442	549,824
Events per Second	4,628.72	14,600	7,607.57
Unique Processes	1,184	546	789
Active Processes	218	123	238
File Operations	410,707	332,144	460,164
IPC Events	85,836	50,680	47,629
Suspicious Patterns	27	8	28
Read/Write Ratio	1.73	1.03	1.39
Most Accessed File Type	.db (7,693)	.db (3,248)	.db (5,302)

Table 5.23: Comparison of Network Metrics and Event Category Distribution

Metric	**Messenger**	**Signal**	**Telegram**
TCP Events	750	263	462
UDP Events	2	0	3
Total TCP Data	3.48 MB	1.62 MB	574.36 MB
UNIX Traffic	2.44 MB sent / 48.2 MB received	None	None
Unique Protocols	4	3	4
Unique Connections	2	1	5
ioctl (%)	46.6	54.4	57.3
binder (%)	18.5	14.4	10.1
read (%)	18.1	13.4	16.8
write (%)	10.5	13.0	12.1
network (%)	6.3	4.8	3.7

When comparing runtime behaviour in the passive background state, a more nuanced picture emerges. Signal demonstrates the most discreet background profile, dropping to near-zero network and IPC activity and performing only lightweight file reads for integrity checks. Messenger shows a moderate background footprint: while its event rate and active processes drop drastically compared to active use, it maintains background file I/O and minimal network pings, likely for push notifications and state synchronisation. Telegram remains the most active in the background among the three: although its event throughput decreases by over 90% relative to its foreground mode, it sustains the highest residual file activity and some low-level network interactions, reflecting its design for persistent cache refresh and background synchronisation.

This comparison, summarised in Table 5.24, confirms that while all three apps reduce processing when idle, Signal aligns most closely with privacy-by-design principles by minimising passive runtime footprint, whereas Telegram trades off background stealth for faster media availability and immediate message sync.

Table 5.24: Background Scenario Comparison Across Applications

Metric	**Messenger (BG)**	**Signal (BG)**	**Telegram (BG)**
Duration (sec)	30.39	29.5	29.83
Total Events	38,965	30,663	39,340
Events per Second	1,283	1,039	1,318
File Operations	38,955	30,658	39,336
IPC Events	0	0	0
TCP Events	8	5	4
Total TCP Data	<0.01 MB	0.02 MB	0.02 MB
Distinctive Patterns	8	8	6
Read/Write Ratio	1.50	168.09	1.45

Table 5.25: Foreground vs Background Comparison for Each Application

Metric	**Messenger (FG/BG)**	**Signal (FG/BG)**	**Telegram (FG/BG)**
Total Events	557,770 / 38,965	423,442 / 30,663	549,824 / 39,340
Events per Second	18,512 / 1,283	14,600 / 1,039	18,625 / 1,318
File Operations	410,707 / 38,955	332,144 / 30,658	460,164 / 39,336
IPC Events	85,836 / 0	50,680 / 0	47,629 / 0
TCP Events	750 / 8	263 / 5	462 / 4
Total TCP Data	3.48 MB / <0.01 MB	1.62 MB / 0.02 MB	574.36 MB* / 0.02 MB
Read/Write Ratio	1.73 / 1.50	1.03 / 168.09	1.39 / 1.45

* Likely artefact; see Section Telegram Foreground Runtime Behaviour".

Together, these updated measurements clearly illustrate how each messaging app adjusts its resource footprint between active user interaction and passive idle states — a crucial aspect that directly impacts privacy preservation, battery life, and perceived responsiveness.

5.2.3 Behavioural Patterns Observed

Messenger Full Foreground Scenario

The Messenger application, when running with full permissions in the foreground, shows an event pattern typical of messaging clients during active usage. As illustrated in the *Event Timeline* (Figure 5.1), the `write` and `ioctl` system calls occur in parallel over a substantial portion of the session, followed by a continuous sequence of `read` operations. Throughout the timeline, `network` activity appears more scattered and intermittent.

This arrangement likely reflects how the application first writes data and performs control operations to manage outgoing content or system resources, then maintains an extended reading phase to handle incoming data streams and message updates. The dispersed network events suggest periodic syncs or background connections rather than a constant data flow.

Overall, this event distribution is representative of real-time messaging behaviour and will be seen in a similar form across different scenarios and applications, highlighting that such patterns stem from the nature of messaging workflows rather than any single app-specific implementation.

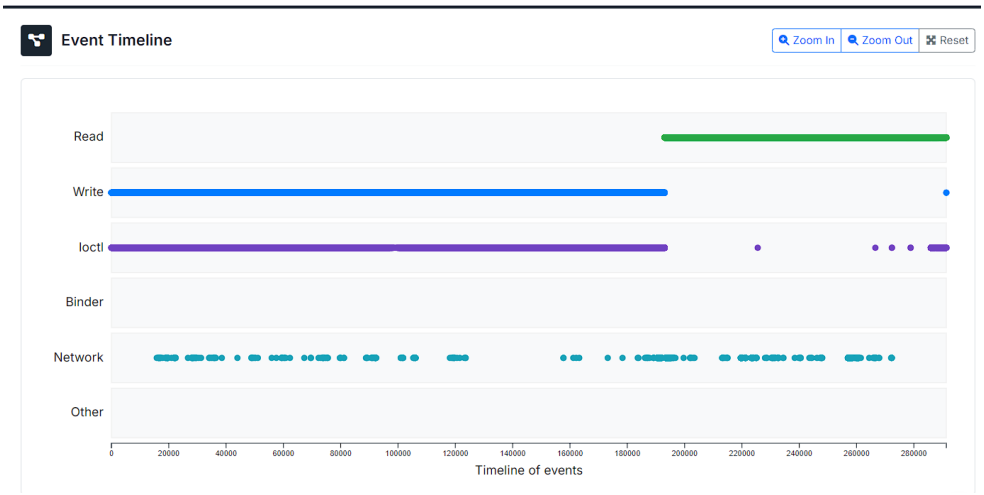

Figure 5.1: Event Timeline of Messenger in Full Foreground Scenario.

The *Behavior Timeline* (Figure 5.2) distinctly highlights Messenger's interaction with user contacts, a behaviour that is largely absent in Signal and only marginally present in Telegram. Here, repeated contact lookups and metadata synchronisations are evident alongside simultaneous camera and audio usage. This confirms Messenger's extensive data integration to support features such as rich chat, stories, and contact suggestions.

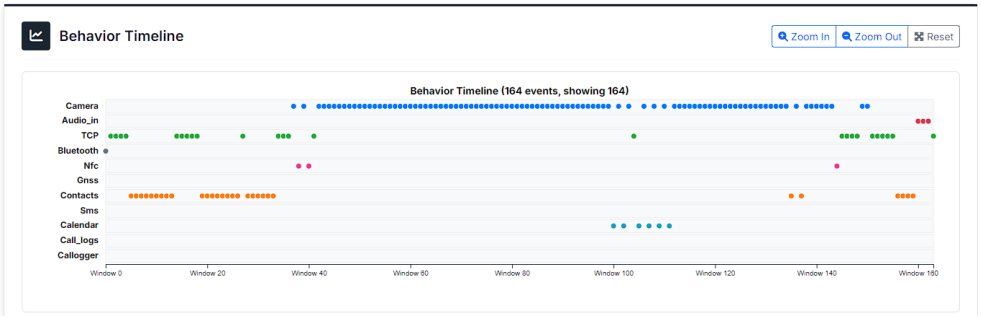

Figure 5.2: Behavior Timeline of Messenger in Full Foreground Scenario.

The *Communication Flow* diagram (Figure 5.3) reinforces the app's architectural complexity: a dense cluster of process IDs branches out from a central dispatcher, with multiple nodes playing a pivotal role in handling high-frequency tasks. Compared to Signal's and Telegram's flows, Messenger maintains more worker PIDs actively participating in both foreground and background operations, highlighting its design to manage simultaneous media, chat, and backend connectivity with minimal latency.

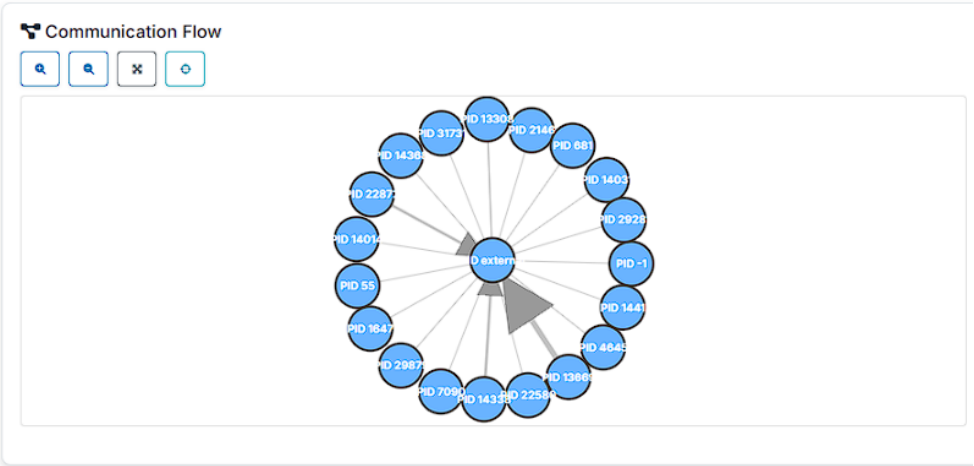

Figure 5.3: Communication Flow of Messenger in Full Foreground Scenario.

Overall, these patterns confirm that Messenger leverages full device permissions not only for multimedia but also for dynamic access to contacts and advanced inter-process orchestration. This expansive behavioural footprint, while supporting richer features, simultaneously broadens the attack surface, which should be considered in any security and privacy evaluation.

Messenger Restricted Permissions Scenario

When executed with restricted permissions, Messenger shows a noticeably smaller runtime footprint compared to its full-permission mode. As shown in the *Event Timeline* in Figure 5.4, the overall pattern remains largely the same as in other scenarios: typical system call streams appear with similar structure and frequency, reflecting how messaging apps handle core I/O and lightweight background checks under similar conditions.

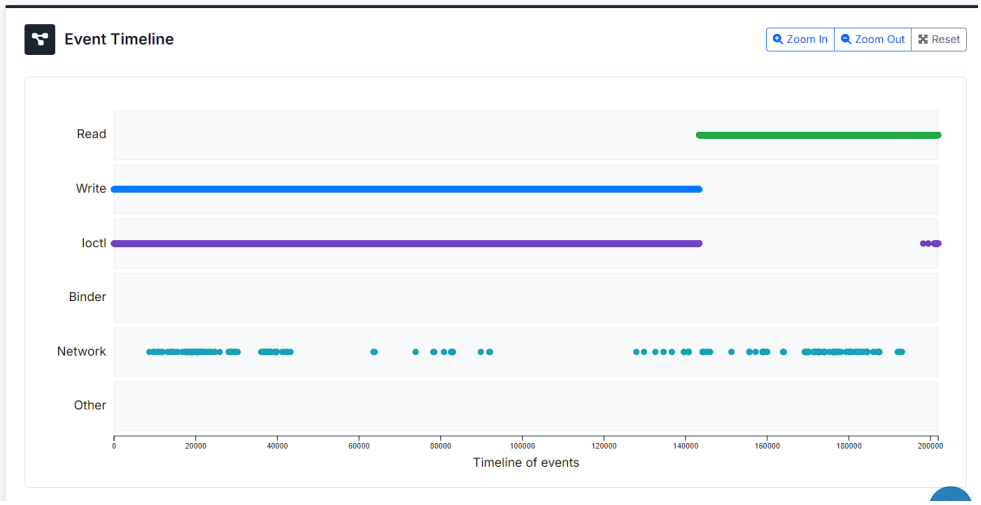

Figure 5.4: Event Timeline of Messenger in Restricted Permissions Scenario.

In contrast, the *Behavior Timeline* (Figure 5.5) reveals a more intriguing pattern. Despite explicit permission restrictions, Messenger appears to access the `contacts` category repeatedly, along with initiating TCP connections. While this observation does not conclusively prove direct access to sensitive contact data, it strongly suggests that certain components related to contact handling may remain active even when permission checks are in place. Given the semantic complexity of tracing at the kernel level, this behavior is examined in greater technical detail in the following paragraphs.

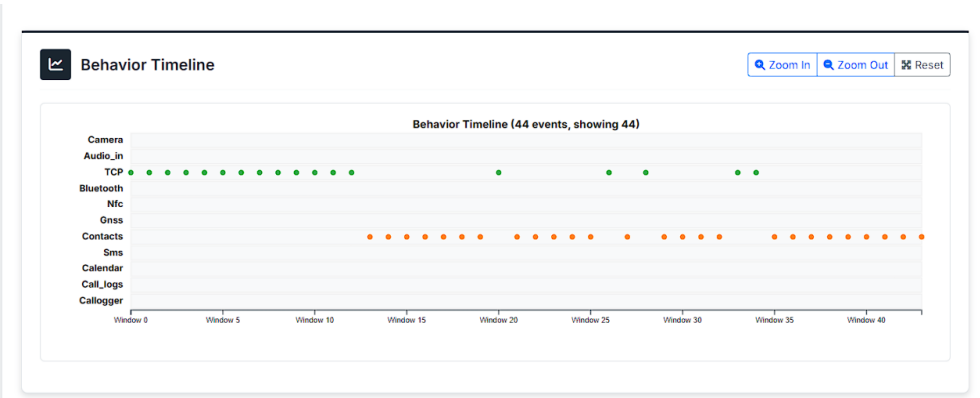

Figure 5.5: Behavior Timeline of Messenger in Restricted Permissions Scenario.

To thoroughly investigate this unexpected behavior, we repeated the same scenario ten times under identical device conditions. The unauthorized contact access appeared consistently in nine out of ten runs, confirming that it is not a random anomaly but rather a reproducible pattern specifically tied to Messenger's runtime behavior. Importantly, detailed kernel-level tracing revealed that the actual access to the contacts database (`contacts2.db`) does not originate directly from the

main Messenger process (`com.facebook.orca`) but instead from a Binder transaction worker thread (`binder:27488_B-7197`). A representative excerpt from the kernel trace clearly illustrates this activity:

```
binder:27488_B-7197 (27488) [000] d..1. 116474.302632: ioctl_probe:
(do_vfs_ioctl+0x0/0xe2c) file=0xffffff808c4597c0 inode=4483 k__dev=0
s_dev_inode=266338357 i_mode=33200 kuid=10064 kgid=10064
pathname="contacts2.db"
```

Despite using both manual analysis and automated information-flow reconstruction through the SliceDroid visualization (*Communication Flow* in Figure 5.6), it was not possible to establish an explicit connection between this Binder worker and the main Messenger app's PID or UID. Nevertheless, repeated tests strongly indicate that this unexpected contact access occurs exclusively when launching Messenger. Similar repeated tests on the other two messaging apps with equally restrictive permissions did not yield the same behavior, further confirming the specificity of this finding to Messenger.

One plausible explanation is that Messenger utilizes indirect IPC mechanisms or legacy code paths, potentially invoking a background system service or Content Provider that bypasses the Android permission enforcement model under certain conditions. However, it is also possible that the observed access is not directly attributable to Messenger itself, but to a third-party component or background process triggered coincidentally during Messenger's launch. The consistent reproduction of access events that appear semantically related to the `contacts` category—exclusively during Messenger sessions—strongly suggests a privacy-relevant behavior that warrants further investigation and responsible remediation by both application developers and the Android platform maintainers.

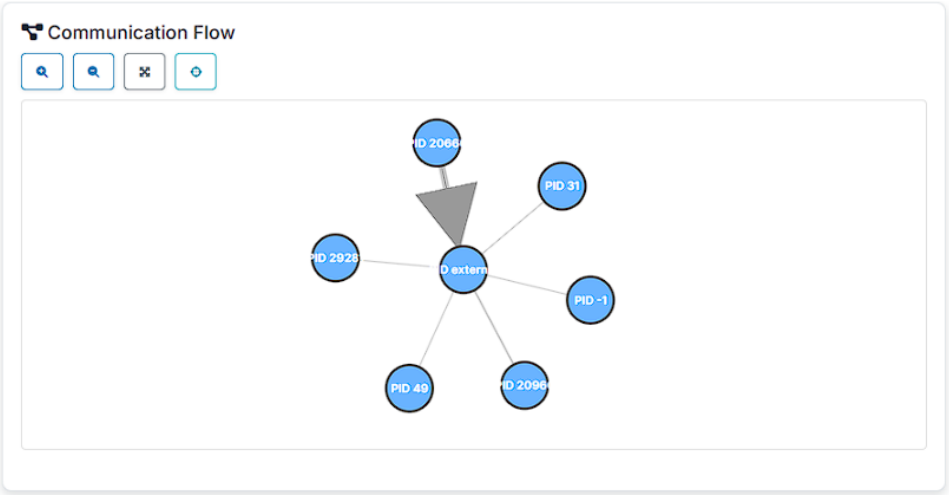

Figure 5.6: Communication Flow of Messenger in Restricted Permissions Scenario.

Overall, while the core event sequence remains consistent across various execution scenarios, the recurring and technically observable indications of access to `contacts2.db` under restricted-permission mode raise a potential privacy concern. Although this thesis does not provide definitive proof of unauthorized data access, the repeated semantic correlation between contact-related activity and Messenger execution constitutes strong evidence that warrants further scrutiny. This finding highlights the need for more robust and transparent permission validation mechanisms within both the Android framework and third-party app implementations, reinforcing the critical importance of behavioral profiling and kernel-level tracing methodologies as demonstrated throughout this work.

Signal Full Foreground Scenario

When running with full permissions, Signal shows similar patterns to other clients.

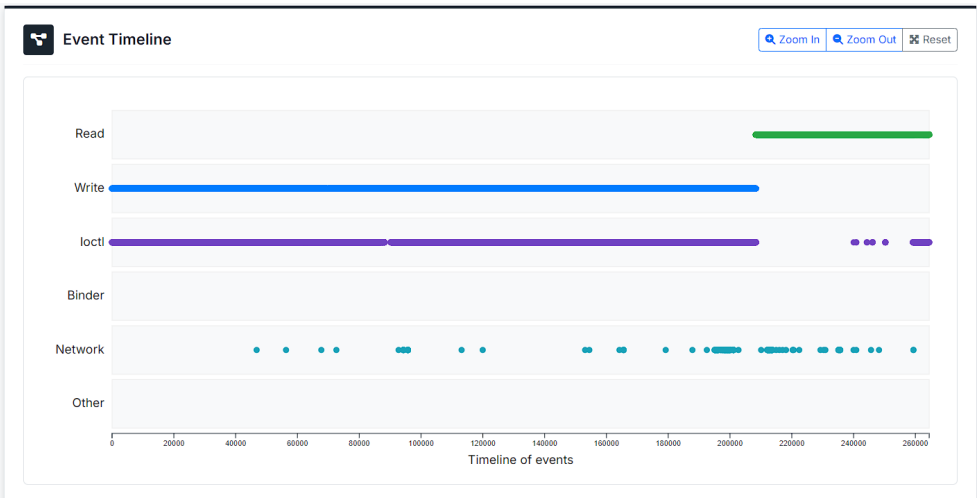

Figure 5.7: Event Timeline of Signal in Full Foreground Scenario.

The *Behavior Timeline* (Figure 5.8) shows consistent camera and microphone use, with TCP connections appearing mostly towards the end of the session for message transmission. Unlike Messenger, no contacts access is observed.

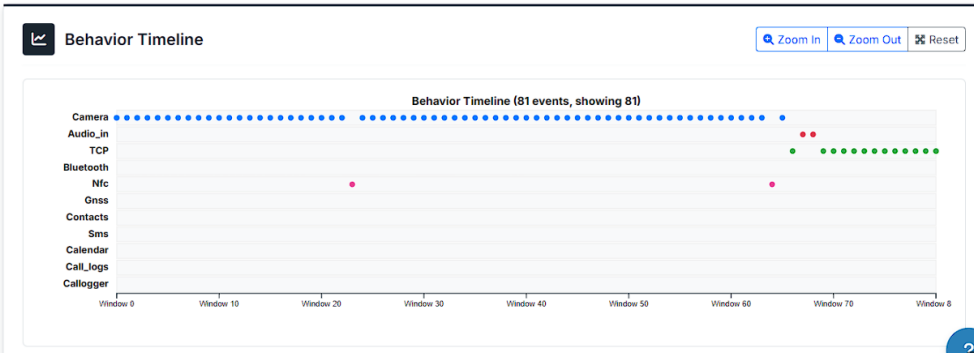

Figure 5.8: Behavior Timeline of Signal in Full Foreground Scenario.

The *Communication Flow* (Figure 5.9) depicts a simple IPC graph with a few worker PIDs, reflecting minimal process spawning and strict sandboxing.

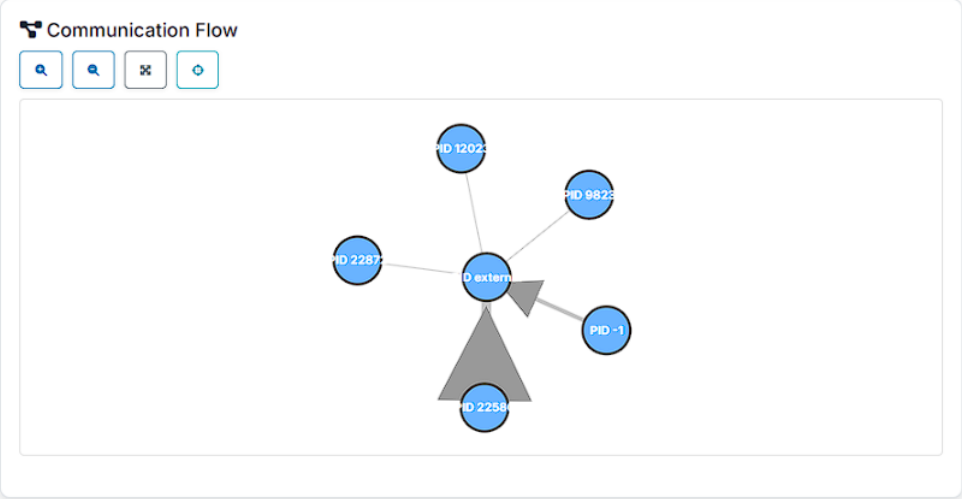

Figure 5.9: Communication Flow of Signal in Full Foreground Scenario.

Overall, these results confirm Signal's design priorities: essential media capture and encrypted transport, with no unnecessary data or contact access.

Signal Restricted Permissions Scenario

When Signal runs with restricted permissions, its runtime behaviour remains simple and predictable. The *Event Timeline* (Figure 5.10) shows the same general pattern as in full mode: `write` and `ioctl` calls occur together initially, followed by a continuous `read` phase, and scattered `network` events.

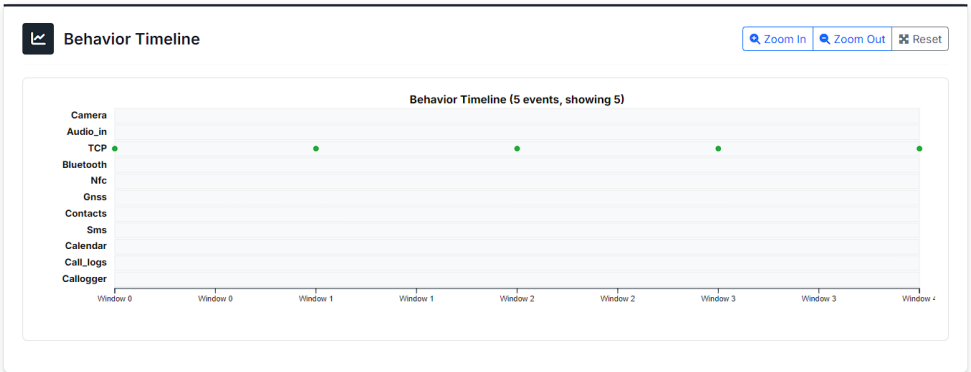

Figure 5.10: Event Timeline of Signal in Restricted Permissions Scenario.

The *Behavior Timeline* (Figure 5.11) confirms minimal behaviour: only a few TCP events appear, with no access to contacts, camera, or microphone, which fully aligns with the restricted permissions granted in this scenario.

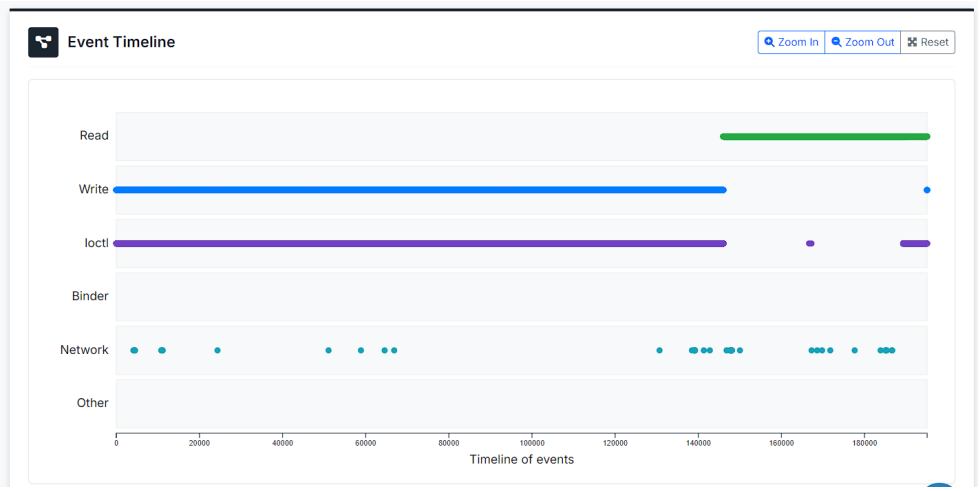

Figure 5.11: Behavior Timeline of Signal in Restricted Permissions Scenario.

Finally, the *Communication Flow* (Figure 5.12) shows an extremely lightweight process graph with just a single child PID, reflecting Signal's strict process isolation and minimal background activity.

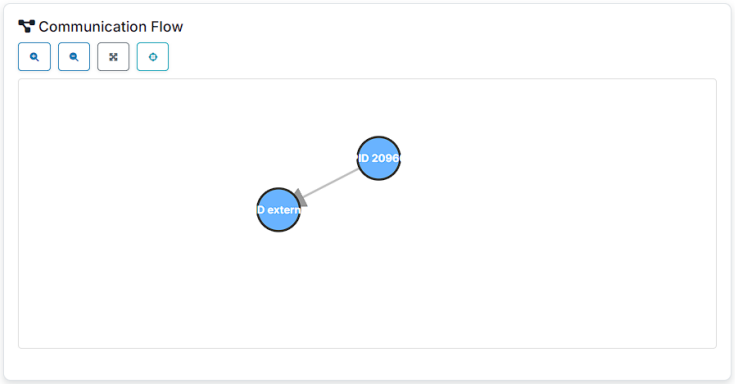

Figure 5.12: Communication Flow of Signal in Restricted Permissions Scenario.

In summary, Signal stays consistent with its design: even under restricted permissions, no sensitive data like contacts is accessed, and only basic network connections are maintained.

Telegram Full Foreground Scenario

When running in full foreground mode, Telegram shows a typical high-activity pattern.

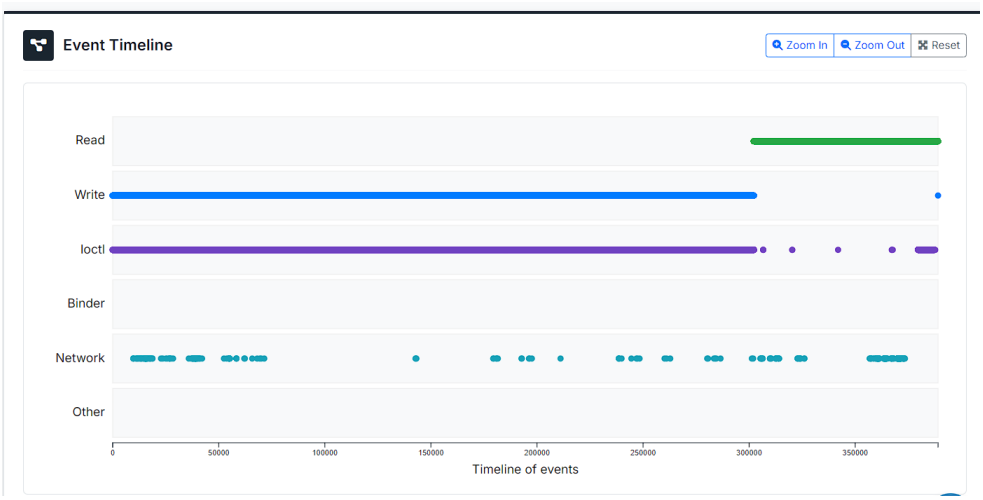

Figure 5.13: Event Timeline of Telegram in Full Foreground Scenario.

The *Behavior Timeline* (Figure 5.14) shows frequent camera and microphone use, stable TCP connections, and clear contact access near the end, indicating background syncing alongside media messaging.

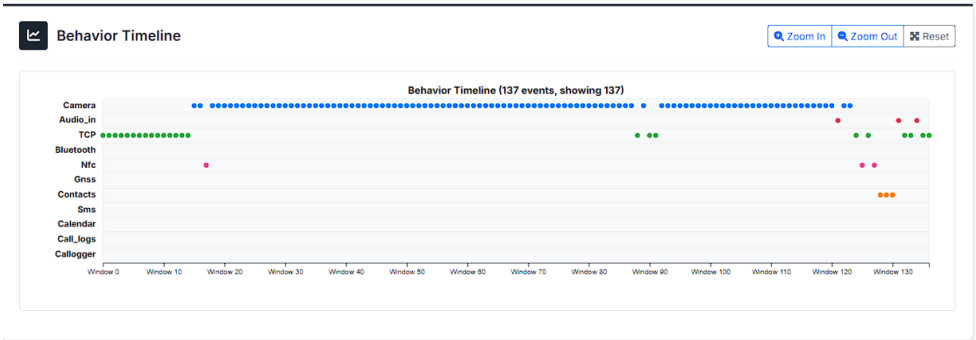

Figure 5.14: Behavior Timeline of Telegram in Full Foreground Scenario.

The *Communication Flow* (Figure 5.15) shows a multi-worker structure similar to Messenger, though slightly less intricate, enabling parallel processing of messages, media, and sync routines.

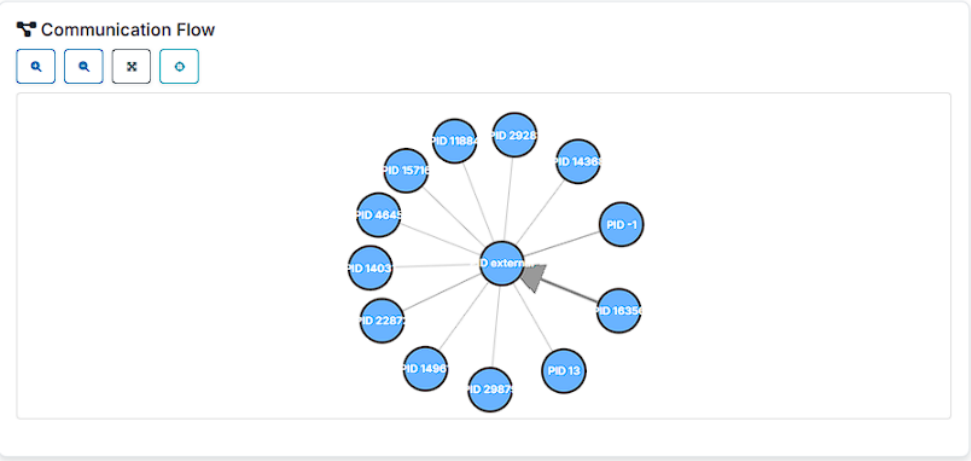

Figure 5.15: Communication Flow of Telegram in Full Foreground Scenario.

Overall, Telegram behaves much like Messenger, offering rich media and contact syncing, but with a slightly simpler and lighter runtime footprint compared to Messenger.

Telegram Restricted Permissions Scenario

When Telegram runs with restricted permissions the *Event Timeline* (Figure 5.16) shows the same general structure as other clients: `write` and `ioctl` appear first, followed by continuous `read` and scattered `network` calls — but all with lower density due to disabled features.

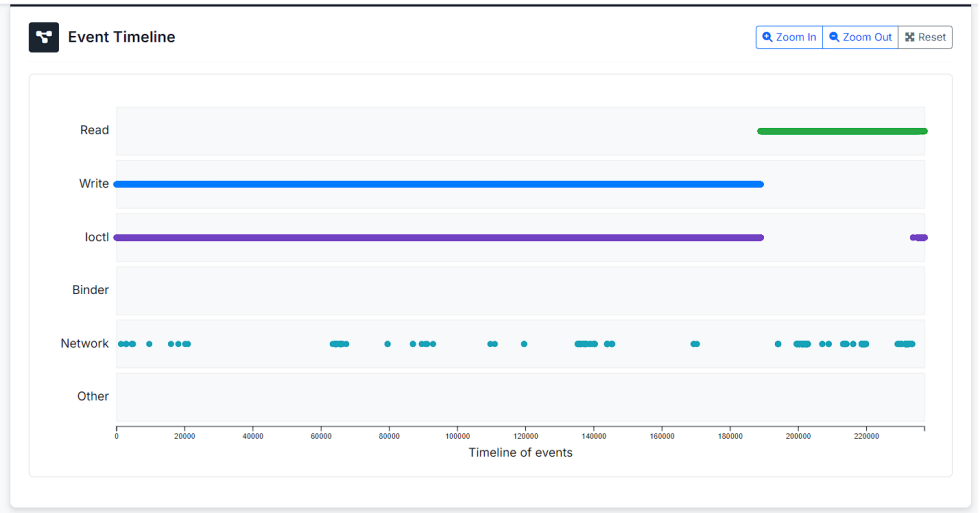

Figure 5.16: Event Timeline of Telegram in Restricted Permissions Scenario.

The *Behavior Timeline* (Figure 5.17) confirms minimal behaviour: only a few TCP events are recorded, while camera, microphone, and contacts remain inactive — fully consistent with the blocked permissions.

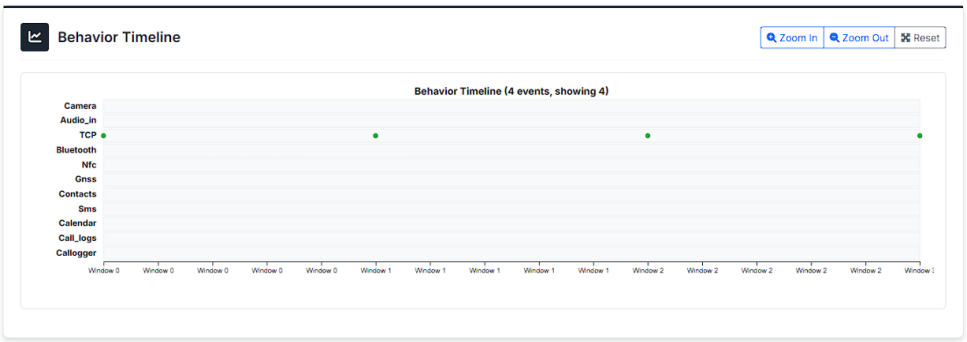

Figure 5.17: Behavior Timeline of Telegram in Restricted Permissions Scenario.

The *Communication Flow* (Figure 5.18) reveals a simple process tree with very few worker PIDs, showing that background media and sync threads are inactive under permission constraints.

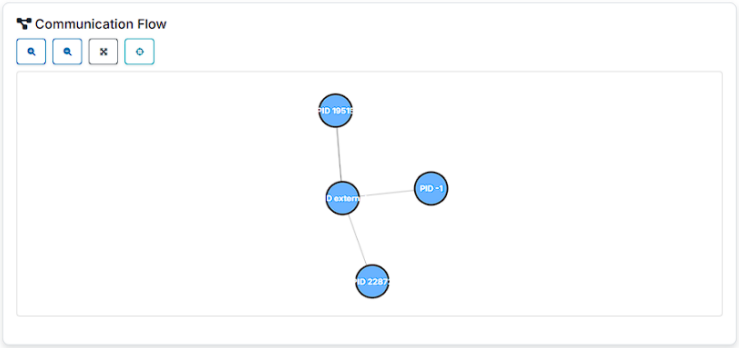

Figure 5.18: Communication Flow of Telegram in Restricted Permissions Scenario.

Overall, Telegram adapts correctly to restricted permissions by falling back to basic network checks only, avoiding sensitive access and unnecessary processing.

5.2.4 Cross-Application Comparative Analysis

To advance beyond descriptive statistics, we conducted a one-way Analysis of Variance (ANOVA) to compare the means of multiple independent groups and test whether the observed differences in application behavior are statistically significant. This section outlines the methodology, assumptions, and post-hoc analyses, providing robust evidence for distinct architectural characteristics supported by inferential testing.

Theoretical Background of ANOVA and Post-Hoc Testing

The validity of ANOVA results hinges on three key assumptions, which were tested prior to analysis:

i. **Normality of Residuals:** The residuals (the differences between individual observations and their group mean) for each group must be normally distributed. This ensures the sampling distribution of the mean is reliable. For an observation Y_{ij} in group i, the errors $\varepsilon_{ij} = Y_{ij} - \mu_i$ should follow a normal distribution:

$$\varepsilon_{ij} \sim \mathcal{N}(0, \sigma^2)$$

ii. **Homogeneity of Variances (Homoscedasticity):** The variance within each group being compared must be equal. This is formally stated by the null hypothesis:

$$H_0 : \sigma_1^2 = \sigma_2^2 = \cdots = \sigma_k^2$$

iii. **Independence of Observations:** The observations in each group, as well as between groups, must be independent of one another.

ANOVA tests the following null and alternative hypotheses concerning the population means μ_i of the k application groups:

i. **Null Hypothesis (H_0):** All group means are equal, implying no difference in the measured metric across applications.

$$H_0 : \mu_1 = \mu_2 = \cdots = \mu_k$$

ii. **Alternative Hypothesis (H_1):** At least one group mean is different, indicating a significant effect of the application choice on the metric.

$$H_1 : \exists i, j \text{ such that } \mu_i \neq \mu_j$$

The F-statistic is calculated as the ratio of the variance between the groups (MS_B) to the variance within the groups (MS_W). A large F-value suggests that the variability between groups is high compared to the variability within groups.

$$F = \frac{MS_B}{MS_W}, \quad \text{where} \quad MS_B = \frac{\sum_{i=1}^{k} n_i(\bar{Y}_i - \bar{Y})^2}{k-1} \quad , \quad MS_W = \frac{\sum_{i=1}^{k}\sum_{j=1}^{n_i}(Y_{ij} - \bar{Y}_i)^2}{N-k}$$

A statistically significant ANOVA result (i.e., rejecting H_0) indicates that a difference exists, but does not specify which groups differ. To identify these specific pairwise differences, we employ a post-hoc test. We selected Tukey's Honestly Significant Difference (HSD) test because it is designed to control the family-wise error rate when performing all possible pairwise comparisons. The HSD critical value is calculated as:

$$HSD = q_{\alpha,k,N-k}\sqrt{\frac{MS_W}{n}}$$

where q is the value from the studentized range distribution for a given significance level α.

Results of Statistical Tests

To ensure the statistical robustness of our analysis and account for performance variability, we conducted 10 independent test runs for each application under controlled conditions. This iterative data collection process yields a more reliable and representative sample, minimizing the influence of anomalous executions or transient system

states. We then applied our statistical methodology to three key performance metrics, calculated from the aggregated data of these 10 runs, to accurately profile each application's typical operational behavior.

(i) Events per Second: This metric measures the rate of system-level activity generated by an application, providing an indication of its overall operational intensity or "chattiness." It is calculated by dividing the total number of monitored system events (e.g., system calls, file access, etc.) by the duration of the measurement period in seconds:

$$R_{\text{Events/sec}} = \frac{\sum N_{\text{Monitored Events}}}{T_{\text{execution (in seconds)}}}$$

A higher value suggests a "busier" application with more frequent background processing or system interaction.

The assumptions of normality and homogeneity of variances were verified, confirming the data's suitability for ANOVA. Specifically, Shapiro-Wilk tests for each application were non-significant (Messenger: $p = 0.1628$; Signal: $p = 0.2032$; Telegram: $p = 0.4871$), as was the Levene's test for equal variances ($p = 0.3892$).

The subsequent ANOVA yielded a statistically significant result ($F(2, N-3) = 9.5461, p = 0.0033$). This indicates a clear difference in the rate of background event generation among the applications. The Tukey HSD post-hoc analysis provided a detailed breakdown:

- *Messenger vs. Signal:* A statistically significant difference was found ($p = 0.0034$).
- *Signal vs. Telegram:* A statistically significant difference was found ($p = 0.0199$).
- *Messenger vs. Telegram:* The difference was **not** statistically significant ($p = 0.5923$).

Interpretation: This result statistically confirms that Signal operates with a significantly lower level of background activity. This aligns with its design philosophy of being a lightweight, resource-conscious application that minimizes its footprint.

(ii) Inter-Process Communication (IPC) Ratio: This metric quantifies the proportion of an application's internal communication overhead relative to its total system-level activity. It is calculated by dividing the number of IPC-related system

calls (e.g., ‘binder‘, ‘socketpair‘, ‘pipe‘) by the total number of events monitored:

$$R_{\text{IPC}} = \frac{\sum N_{\text{IPC Events}}}{\sum N_{\text{Total System Events}}}$$

A high value for this ratio indicates that a significant fraction of the application's work is dedicated to communication between its distinct components, which is characteristic of a modular or multi-process architecture.

For this metric, the normality assumption was violated for the Messenger application. Therefore, the ANOVA results should be interpreted with caution. Nonetheless, the test produced an extremely strong F-statistic ($F(2, N-3) = 35.6129, p < 0.001$). The subsequent Tukey HSD test indicated that **all pairwise comparisons** were highly significant.

Interpretation: Messenger's exceptionally high IPC ratio strongly suggests a complex, modular architecture. This design relies heavily on communication between distinct processes or services to manage its rich feature set, a common architectural pattern in large-scale, mature applications.

(iii) Network-to-File Ratio: This ratio directly compares an application's network data dependency against its reliance on local file storage. It is calculated as the ratio of total bytes transferred over network sockets to the total bytes read from and written to the local filesystem:

$$R_{\text{Net/File}} = \frac{\sum B_{\text{Network I/O}}}{\sum B_{\text{File I/O}}}$$

This metric effectively reveals an application's core operational strategy. A high ratio ($\gg 1$) signifies a "cloud-native" design that streams data on-demand, whereas a low ratio ($\ll 1$) points to an architecture that relies heavily on local caching and offline data access.

All ANOVA assumptions were satisfied for this metric. The analysis revealed a highly significant difference among the applications ($F(2, N-3) = 19.9452, p = 0.0002$). The Tukey HSD test clarified that Messenger's ratio is significantly different from both Signal and Telegram, while the latter two are not significantly different from each other on this metric.

Interpretation: Telegram's distinctively high ratio points to a cloud-native architectural model. The application is optimized for high-volume network data transfers (e.g., fetching media and extensive chat histories from the cloud) relative to its local file I/O operations, clearly distinguishing its operational strategy from the other applications.

Conclusive Synthesis and Interpretation

The collective results of these statistical tests provide quantitative, verifiable proof of the architectural trade-offs inherent in each application's design. The analysis moves beyond simple observation to confirm the following data-driven profiles:

- **Messenger** is characterized as a feature-rich, complex ecosystem that leverages a high degree of Inter-Process Communication to orchestrate its many functionalities.
- **Signal** is statistically verified as a resource-efficient application, deliberately engineered to minimize its background footprint by generating significantly fewer system events.
- **Telegram** exhibits a cloud-centric architecture, prioritizing high-throughput network operations to deliver a seamless, cloud-synchronized user experience, setting it apart from more device-centric models.

These findings successfully align with our thesis goals, providing a robust statistical foundation for understanding the different design philosophies and performance footprints of these widely-used messaging applications. A summary of the p-values is provided in Table 5.26.

Table 5.26: Summary of ANOVA Test Results and p-values

Performance Metric	**p-value**
Events per Second	0.0033
IPC Ratio	¡ 0.001 (cautionary)
Network-to-File Ratio	0.0002

Chapter 6

Discussion

6.1 Interpretation of Results

The combined static and dynamic analyses yield a nuanced understanding of the architectural and operational trade-offs embodied by *Signal*, *Telegram*, and *Messenger*. Our findings confirm the central hypothesis: that each application's security posture and runtime behaviour are direct reflections of its underlying design principles. This aligns with broader research demonstrating that an app's privacy implications are deeply rooted in its architecture and choice of third-party libraries [39].

From a static perspective, Signal's lean manifest configuration and restrictive permission set correspond with its security-first design, a finding consistent with formal analyses of its protocol that highlight its cryptographic robustness [10]. In contrast, Messenger exhibits the broadest architectural footprint, featuring an extensive inventory of components and third-party telemetry SDKs. This observation corroborates large-scale studies by Shen et al., who identified Facebook's infrastructure as a central hub for data collection via embedded trackers [39]. Telegram's architecture presents a middle ground; while free of third-party trackers, its allowance of clear-text HTTP traffic and extensive permissions align with forensic studies that have noted its hybrid security model, which prioritises features like cloud synchronisation over default end-to-end encryption [30].

Dynamic tracing validates these static insights by providing empirical evidence of runtime behaviour. The finding that Messenger generates the highest event throughput and most intensive inter-process communication is consistent with its complex, feature-rich design. Signal's minimal runtime footprint reinforces its characterisation as a privacy-preserving application, while Telegram's high network-to-file ratio underscores its cloud-native architecture.

Perhaps the most critical finding is the observation of Messenger's access to the

`contacts2.db` database, even when the explicit runtime permission is denied. This exemplifies the "permission gap" in Android, where privacy leaks can occur not through direct permission abuse, but via indirect inter-component communication (ICC) channels [23]. While our kernel-level approach cannot definitively trace the IPC chain, it provides strong evidence of an interaction that bypasses the user-facing permission model. This aligns with research by Lu et al. on component hijacking [15] and reinforces the value of kernel-level observability for uncovering behaviours that are invisible to both static analysis and user-space instrumentation.

Finally, the statistically significant differences in event rates and resource usage patterns between the applications, confirmed by ANOVA and Tukey HSD tests, provide quantitative validation for these qualitative distinctions. While other studies have performed forensic analyses of these apps at rest [42, 34], our work contributes by delivering a comparative analysis of their live system behaviour, demonstrating that architectural philosophies translate into measurable, kernel-level operational fingerprints.

6.2 Answers to Research Questions

RQ1. What systematic methodology can be used for a comparative security and privacy analysis of mobile messaging applications? This study demonstrates that a robust comparative methodology should integrate static analysis (using tools such as MobSF, APKID, and Androguard) with dynamic kernel-level tracing. Static inspection reveals manifest configurations, declared permissions, and embedded secrets, while dynamic tracing uncovers real runtime behaviour, including syscall patterns, inter-process communication, and actual network usage. Together, these methods provide a replicable, data-driven framework for assessing mobile application privacy and security.

RQ2. Is static analysis alone sufficient to uncover privacy-intrusive behaviour, or do kernel-level traces reveal additional hidden operations and syscall patterns? The results suggest that static analysis alone may not be sufficient to uncover the full range of privacy-relevant behaviours. Kernel-level traces exposed runtime activity that appears inconsistent with manifest-level declarations — for instance, Messenger seemed to interact with contacts-related resources even when permissions were explicitly restricted. While this observation does not conclusively prove unauthorized access, it highlights behaviours that are not visible through static inspection alone. Additionally, dynamic analysis uncovered characteristics such

as background synchronization and large-scale media prefetching, which would remain undetected by static tools. Therefore, the combination of static and dynamic methods provides a more comprehensive understanding of potential privacy risks.

RQ3. Which system calls do popular messaging apps issue during normal use, and how do these calls align with—or diverge from—their declared permissions? The analysis confirms that all three applications rely heavily on standard system calls such as `ioctl`, `read`, `write`, and `binder` for routine operation. Generally, the syscall usage aligns with the permissions declared in their manifests. However, dynamic traces exposed cases where behaviour diverged: Messenger's continued contact access under restricted permissions illustrates a gap between static declarations and runtime execution, highlighting the value of tracing for verifying real adherence to permission models.

RQ4. How do privacy-centric apps (e.g., Signal) compare to mainstream platforms in terms of data minimisation, syscall footprint, and overall privacy posture? Signal exhibits the leanest runtime signature, the lowest event throughput, and the smallest network footprint among the three apps, all of which validate its privacy-first architecture. In contrast, Messenger shows high inter-process complexity and extensive local data operations, supporting its feature-rich functionality but broadening its attack surface. Telegram emphasises a network-heavy design, evident in its high network-to-file ratio and large data transfers, which enable fast media delivery but increase dependency on secure cloud infrastructure. These statistically validated profiles demonstrate distinct trade-offs between feature richness and privacy preservation.

In summary, this thesis confirms that static and dynamic analyses, combined with inferential statistical validation, provide a robust, replicable basis for assessing the security and privacy posture of mobile messaging applications in practice.

6.3 Opportunities for Improvement

Many of the opportunities for future enhancement directly address the methodological and project-specific constraints outlined in Section 1.4. This thesis demonstrates the value of combining static and dynamic analysis, but several areas remain for significant enhancement. The primary opportunity lies in expanding the experimental scope. Future work should conduct a large-scale analysis across a more diverse set of applications from various categories [26] and extend the trace duration from seconds

to minutes or even hours. Such a longitudinal approach would capture subtle background and idle-state behaviors often missed in short-term observations [37]. This would also allow for profiling more complex user interaction scenarios (e.g., group video calls, large file transfers) that could expose different system behaviours and protocol usage, such as UDP. Notably, further examination of the `contacts2.db` access by Messenger—possibly via dynamic taint analysis [16]—could determine whether this is due to indirect IPC or a platform-level oversight.

On the tooling side, integrating deeper network-layer visibility would enhance behavioral reconstruction. Beyond capturing packet data with `tcpdump` [26], this could include DNS query analysis to identify server endpoints and inspection of TLS handshake metadata (e.g., Server Name Indication - SNI), which reveals communication endpoints even through encrypted traffic [1]. Furthermore, improving trace labelling automation to reconstruct API semantics from syscalls [33] and expanding the analysis to include pre-installed system apps [43] would offer a more holistic perspective on mobile device behaviour.

Chapter 7

Conclusions

7.1 Key Findings

This thesis set out to systematically assess the security and privacy posture of three widely used mobile messaging applications—Signal, Telegram, and Messenger—by combining rigorous static configuration inspection with low-level kernel-based dynamic tracing. The findings confirm that each application embodies distinct architectural trade-offs that manifest clearly in both static risk factors and runtime system behaviour. Static analysis revealed that Signal consistently maintains the leanest manifest and permission configuration, with minimal exported components and no bundled trackers, aligning closely with its privacy-first design philosophy. In contrast, Messenger exhibits the largest architectural footprint, with a complex component graph, broad permission set, and inclusion of third-party telemetry SDKs, reflecting a design optimised for feature richness at the expense of a larger attack surface. Telegram situates itself between these extremes: it avoids trackers but adopts a more permissive network configuration and permits clear-text traffic.

Dynamic tracing validated and extended these insights by quantifying actual system call patterns under controlled usage scenarios. Signal demonstrated the smallest event throughput and background footprint, confirming its resource-conscious approach. Messenger generated the highest volume of inter-process communication and file I/O, evidencing a rich but complex orchestration of services. Telegram displayed a network-dominant profile, with a significantly higher network-to-file ratio, indicating its cloud-native strategy for fast media synchronisation. Inferential statistical tests confirmed that these differences are not incidental but statistically significant, lending robust support to the core research questions. In summary, the combined analysis illustrates that while all three applications meet baseline security expectations, their operational behaviours differ in ways that directly reflect their respective

design priorities, impacting user privacy and resource consumption.

Among the most notable runtime observations was Messenger's apparent access to contact-related resources—even when explicit runtime permissions had been denied. While this does not constitute definitive proof of unauthorized access, repeated kernel-level traces consistently revealed Binder transactions involving the contacts2.db file—behaviour not observed in Signal or Telegram. This pattern, unique to Messenger across all controlled experiments, underscores the importance of dynamic tracing in identifying semantically relevant system-level activity that may elude static inspection. The precise origin of these accesses remains inconclusive; however, their consistency highlights a privacy-relevant anomaly that warrants further investigation.

7.2 Future Work

The methodological framework established in this thesis opens several avenues for future research, moving beyond the immediate technical details to address broader questions of digital privacy and security.

A primary next step is to apply this combined analytical approach to a wider and more diverse range of mobile applications. While this study focused on messaging apps, the methodology could yield significant insights into other domains where user data is sensitive, such as banking, healthcare (mHealth), or social media applications. Such a large-scale study would help establish a broader baseline for "normal" versus "suspicious" behaviour across the mobile ecosystem and could reveal systemic privacy risks tied to specific app categories or development practices.

Furthermore, future work should investigate the long-term behaviour of applications. The current analysis captured short, controlled user interactions. Extending the observation period to hours or even days would allow for a more comprehensive understanding of an app's lifecycle, particularly its idle-state and background activities. This longitudinal analysis could uncover subtle, periodic data collection or communication patterns that are invisible during brief tests but are critical to an application's overall privacy footprint.

Finally, a crucial direction is to conclusively identify the source of privacy-anomalous behaviours, such as Messenger's access to the contacts database. While this study provided strong evidence of the behaviour at the kernel level, the next logical step would be to employ more advanced techniques to trace the exact chain of events within the application that leads to this access. Answering such questions would not only solve a technical puzzle but also provide key insights into bypassing platform security models, helping developers and vendors build more secure systems.

Bibliography

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

[2] D. Arp, M. Spreitzenbarth, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of Android malware in your pocket," in *Proc. NDSS '14*, 2014.

[3] S. Arzt *et al.*, "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps," in *Proc. PLDI '14*, 2014.

[4] T. Berezowski, "Push Notification Leakage in Mobile IM Apps," in *Proc. ACSAC WIP '20*, 2020.

[5] J. Bock *et al.*, "A Formal Analysis of Signal's Sealed-Sender," in *Proc. NDSS '20*, 2020.

[6] 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.

[7] Y. Zheng *et al.*, "C-Android: Container-based dynamic analysis for Android," *Journal of Information Security and Applications*, vol. 47, pp. 293-304, 2019.

[8] G. Celik and V. Gligor, "Kernel-Level Tracing for Detecting Stegomalware and Covert Channels," *Computer Networks*, vol. 193, 2021.

[9] S. J. Oishwee, Z. Codabux, and N. Stakhanova, "Decoding Android Permissions: A Study of Developer Challenges and Solutions on Stack Overflow," in *Proc. 18th ACM/IEEE Int. Symp. on Empirical Software Engineering and Measurement (ESEM '24)*, Barcelona, Spain, Oct. 2024, pp. 143–153, doi:10.1145/3674805.3686676.

[10] K. Cohn-Gordon, C. Cremers, B. Dowling, L. Garratt, and D. Stebila, "A Formal Security Analysis of the Signal Messaging Protocol," in *Proc. of the IEEE*

European Symposium on Security and Privacy (EuroS&P), Paris, France, Apr. 2017.

[11] H. González *et al.*, "DYNAMO: Differential dynamic analysis of the Android framework," in *Proc. NDSS '21*, 2021.

[12] M. Nasir *et al.*, "DynaLog: A dynamic analysis framework for Android applications," *arXiv preprint arXiv:1603.02358*, 2016.

[13] T. Sutter, T. Kehrer, M. Rennhard, B. Tellenbach, and J. Klein, "Dynamic Security Analysis on Android: A Systematic Literature Review," *IEEE Access*, vol. 12, pp. 9234-9262, 2024. [Online]. Available: `https://doi.org/10.1109/ACCESS.2024.3357593`

[14] M. Egele, "Persistence of Deleted Media in Telegram CDNs," in *Proc. IMC '19*, 2019.

[15] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: Statically vetting Android apps for component hijacking vulnerabilities," in *Proc. of the 19th ACM conference on Computer and communications security (CCS '12)*, 2012, pp. 260-271.

[16] W. Enck *et al.*, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, article 5, 2014.

[17] A. P. Felt *et al.*, "Android Permissions Demystified," in *Proc. 18th ACM Conference on Computer and Communications Security (CCS)*, 2011.

[18] 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.

[19] S. Frolov, "GDPR Compliance in Mobile Messaging Apps," in *Proc. PETS '22*, 2022.

[20] K. Papadopoulos *et al.*, "InviSeal: Stealthy instrumentation for Android," in *Proc. AsiaCCS '23*, 2023.

[21] D. Kwon, "Triangulating Telegram Users via 'People Nearby'," in *Proc. IEEE S&P '21*, 2021.

[22] H. Lee, "QUIC-based Traffic Fingerprinting of Messaging Apps," in *Proc. TMA '23*, 2023.

[23] L. Li, A. Bartel, T. F. Bissyandé, J. Klein *et al.*, "IccTA: Detecting inter-component privacy leaks in Android apps," in *Proc. ICSE '15*, 2015.

[24] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein, "DroidRA: Taming reflection to support whole-program analysis of Android apps," in *Proc. ISSTA '16*, 2016.

[25] S. Li, Z. Qian, and F. Zhang, "LibDroid: Summarizing information flow of Android native libraries via static analysis," *Digital Investigation*, vol. 40, 2022.

[26] M. Lindorfer *et al.*, "ANDRUBIS - 1,000,000 Apps Later: A View on Current Android Malware Behaviors," in *Proc. BADGERS '14*, 2014.

[27] C. Marforio, "SGX Side-Channels on Private Set Intersection," in *Proc. ASI-ACCS '22*, 2022.

[28] S. Matic *et al.*, "iMessage Privacy," in *Proc. USENIX Security '15*, 2015.

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

[30] A. Moltchanov *et al.*, "Telegram—A Forensic View," in *Proc. DFRWS EU '18*, 2018.

[31] A. R. Nasser, A. M. Hasan, and A. J. Humaidi, "DL-AMDet: Deep learning-based malware detector for Android," *Intelligent Systems with Applications*, vol. 21, 2024.

[32] Z. Ning and F. Zhang, "DexLEGO: Reassembleable bytecode extraction for aiding static analysis," in *Proc. DSN '18*, 2018.

[33] S. Nisi *et al.*, "Reconstructing API semantics from syscall traces on Android," in *Proc. RAID '19*, 2019.

[34] S. Obermeier and S. Diederich, "Signal forensics on Android devices," *Journal of Digital Forensics, Security and Law*, vol. 13, no. 2, 2018.

[35] D. Octeau, P. McDaniel, S. Jha *et al.*, "Effective inter-component communication mapping in Android with Epicc," in *Proc. USENIX Security '13*, 2013.

[36] J. Poblete, "Adaptive Padding against IM Traffic Analysis," *Computer Communications*, vol. 178, pp. 104-113, 2021.

[37] A. Razaghpanah, R. Nithyanand, N. Vallina-Rodriguez, S. Sundaresan, M. Allman, C. Kreibich, and P. Gill, "Apps, Trackers, Privacy, and Regulators: A Global Study of the Mobile Tracking Ecosystem," in *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[38] A. Schmidt, H. Schmidt, J. Clausen, A. M. T. K. Camtepe, and S. Albayrak, "Static analysis of executables for collaborative malware detection on Android," in *Proc. of the IEEE International Conference on Communications (ICC)*, 2009.

[39] Y. Shen, P.-A. Vervier, and G. Stringhini, "Understanding worldwide private information collection on Android," *arXiv preprint arXiv:2102.12869*, 2021.

[40] A. R. Onik, J. Brown, C. Walker, and I. Baggili, "A Systematic Literature Review of Secure Instant Messaging Applications from a Digital Forensics Perspective," *ACM Computing Surveys*, vol. 57, no. 9, Article 239, pp. 1–36, May 2025. [Online]. Available: `https://doi.org/10.1145/3727641`

[41] P. Tang, "Hash-Based Enumeration Attack on WhatsApp Contact Discovery," in *Proc. CCS '20*, 2020.

[42] C. Anglano, "Forensic Analysis of WhatsApp Messenger on Android Smartphones," *arXiv preprint arXiv:1507.07739*, 2015.

[43] J. Gamba, M. Rashed, A. Razaghpanah, J. Tapiador, and N. Vallina-Rodriguez, "An Analysis of Pre-installed Android Software," in *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2020.

[44] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data-flow analysis framework for security vetting of Android apps," in *Proc. CCS '14*, 2014.

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

[46] F. Zhang *et al.*, "HART: Hardware-Assisted Kernel Module Tracing on ARM," in *Proc. ESORICS '20*, 2020.

[47] M. Zheng and M. Sun, "DroidTrace: Ptrace-based dynamic syscall tracing for Android," *International Journal of Information Security*, vol. 13, no. 4, pp. 359-370, 2014.

[48] 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/

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

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

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

[52] 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

[53] R. Thomas, "QBDI and DBI on ARM," in *BlackHat Asia*, 2020.

[54] M. S. Sibal and N. S. Roshni, "Signal, Telegram see demand spike as new WhatsApp terms stir debate," *Reuters*, Jan. 2021. [Online]. Available: https://www.reuters.com/article/idUSKBN29E04O/

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

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

[57] Statista, "Leading mobile messaging apps worldwide as of June 2024, by number of monthly active users," 2024. [Online]. Available: https://www.statista.com/statistics/258749/most-popular-global-mobile-messenger-apps/

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

[59] Statista, "Signal - monthly active users in selected countries 2021," 2022. [Online]. Available: https://www.statista.com/statistics/1252396/signal-monthly-active-users-in-selected-countries/

[60] Statista, "Number of Telegram monthly active users worldwide from 2018 to 2024," 2024. [Online]. Available: https://www.statista.com/statistics/1242337/telegram-users/

[61] The Washington Post, "Pentagon officials used Signal messaging app, raising security concerns," Mar. 2023.

[62] 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/

[63] A. Desnos and contributors, "Androguard: Reverse engineering and analysis of Android applications," 2023. [Online]. Available: https://github.com/androguard/androguard

[64] eBPF Foundation, "eBPF: Extended Berkeley Packet Filter," 2023. [Online]. Available: https://ebpf.io

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

[66] skylot, "Jadx: Dex to Java decompiler," [Online]. Available: https://github.com/skylot/jadx

[67] A. Abraham, "Mobile Security Framework (MobSF)," 2023. [Online]. Available: https://github.com/MobSF/Mobile-Security-Framework-MobSF

[68] The strace developers, "strace: Diagnostic, debugging and instructional userspace utility for Linux," 2023. [Online]. Available: https://strace.io

[69] rovo89, "Xposed Framework," [Online]. Available: https://github.com/rovo89/Xposed

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

[71] Android Open Source Project, "Platform Architecture," 2024. [Online]. Available: https://source.android.com/docs/core/architecture

[72] Android Open Source Project, "Android Runtime (ART) and Dalvik," 2024. [Online]. Available: https://source.android.com/docs/core/dalvik

[73] Android Open Source Project, "Generic Kernel Image (GKI)," 2024. [Online]. Available: https://source.android.com/docs/core/gki

[74] Android Open Source Project, "JNI tips," 2024. [Online]. Available: https://developer.android.com/training/articles/perf-jni

[75] Android Open Source Project, "Android Security Overview," 2024. [Online]. Available: https://source.android.com/docs/security/overview/android

[76] Android Open Source Project, "PowerManager.WakeLock," 2024. [Online]. Available: https://developer.android.com/reference/android/os/PowerManager.WakeLock

[77] The Linux Kernel, "debugfs," The Linux Kernel Documentation. [Online]. Available: https://www.kernel.org/doc/html/latest/filesystems/debugfs.html

[78] S. Rostedt, "ftrace: The Linux Kernel Function Tracer," The Linux Kernel Documentation, 2023. [Online]. Available: https://www.kernel.org/doc/html/latest/trace/ftrace.html

[79] The Linux Kernel, "Kernel Probes (Kprobes)," The Linux Kernel Documentation, 2023. [Online]. Available: https://www.kernel.org/doc/html/latest/trace/kprobes.html

[80] R. Love, *Linux Kernel Development*, 3rd ed., Addison-Wesley, 2010.

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

[82] European Parliament and Council, "Regulation (EU) 2016/679 (General Data Protection Regulation)," *Official Journal of the European Union*, L119, May 2016.

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

Appendix A

Appendix: Code

App Mapping Main Logic

The following excerpt shows the core logic responsible for generating the app mapping by pulling APKs from a connected Android device and analyzing them to extract commercial names using `androguard`.

```
def get_app_label(self, package_name: str) -> Optional[str]:
    if not ANDROGUARD_AVAILABLE:
        return None
    try:
        result = subprocess.run(["adb", "shell", "pm", "path", package_name
            ],
                                capture_output=True, text=True, timeout=10)
        for line in result.stdout.splitlines():
            if line.startswith("package:"):
                device_apk_path = line.split(":", 1)[1].strip()
                break
        temp_apk = tempfile.NamedTemporaryFile(suffix=".apk", delete=False)
        subprocess.run(["adb", "pull", device_apk_path, temp_apk.name],
                       capture_output=True, text=True, timeout=30)
        apk = APK(temp_apk.name)
        return apk.get_app_name()
    except Exception:
        return None
    finally:
        if os.path.exists(temp_apk.name):
            os.unlink(temp_apk.name)
```

```
def create_mapping(self, limit: int = 30, include_system: bool = False) ->
    Dict[str, Dict]:
    mapping = {}
    installed_packages = self.get_installed_packages(user_apps_only=True)
    for package in installed_packages:
        name = self.get_app_label(package)
        if name:
            mapping[package] = {
                "package_name": package,
                "commercial_name": name,
                "processes": [package[-15:]],
                "is_running": True
            }
    return mapping
```