Athens University of Economics and Business

Department of Management Science and Technology

Undergraduate Thesis

Behavioral Profiling of Popular Messaging Apps Using Kernel-Level Tracing

Student:

Foivos - Timotheos Proestakis

Student ID: 8210126

Supervisors:

Prof. Diomidis Spinellis

Dr. Nikolaos Alexopoulos

Submission Date:

May 21, 2025

Abstract

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

Acknowledgments

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

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

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

Contents

Chapter 1

Introduction

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

Among these applications, messaging platforms such as WhatsApp, Telegram, Facebook Messenger, and Signal have gained significant popularity, playing a central role in both personal and professional communication. However, the ubiquitous use of smartphones for such purposes has led to the accumulation of sensitive personal data on user devices, including photos, contact lists, location history, and financial information, thereby raising serious privacy and security concerns [3].

Incidents such as Facebook's unauthorized collection of SMS texts and call logs from Android devices [3] 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 [4, 5].

Despite these legislative efforts, Android's current permission management system remains insufficient. Users frequently misinterpret the scope and implications of the permissions they grant, inadvertently exposing sensitive data to misuse [6, 7].

To address these challenges, it is essential to analyze application behavior—that is, the actual operations performed by an app, both in the foreground and background. Research has shown that discrepancies often exist between user expectations and actual app behavior, with applications executing hidden or unauthorized tasks [**?**, 8]. Many detection techniques rely on the assumption that user interface (UI) elements accurately represent application functionality, an assumption that is not always valid [9].

Behavioral analysis methods are typically divided into static and dynamic approaches. Static analysis examines application code without execution, identifying known malicious patterns. However, it is susceptible to evasion through obfuscation and polymorphism [23, 11]. In contrast, dynamic analysis evaluates applications during runtime, monitoring behaviors such as system calls, resource consumption, and network activity [12, 13]. Among these, system call analysis is particularly valuable, offering fine-grained visibility into application interactions with hardware and OS-level services [14].

Kernel-level tracing is a powerful form of dynamic analysis, capable of capturing low-level system interactions with high precision. Android is built on a modified Linux kernel that orchestrates resource management and system processes via system calls [15]. Tools such as `ftrace` and `kprobes` enable developers and researchers to trace kernel-level function calls, execution flows, and resource usage [16, 17].

`Ftrace` is a built-in tracing utility within the Linux kernel, optimized for performance and capable of monitoring execution latency and function call sequences. `Kprobes`, on the other hand, allows for dynamic instrumentation of running kernels, enabling targeted probing of specific code locations during runtime [18].

Applying kernel-level tracing to messaging applications, however, introduces unique technical challenges. These apps typically exhibit complex, multi-threaded behavior, frequent background processing, and diverse interactions with system resources. Accurately profiling such behavior requires collecting and interpreting high-volume, high-resolution kernel data [19, 20].

Despite the growing research interest in Android security and behavioral analysis, existing work has primarily focused on general application profiling or malware detection. Few studies have concentrated specifically on behavioral profiling of messaging apps using kernel-level data [21]. Meanwhile, recent reports concerning the usage of secure messaging apps such as Signal by government and military officials have emphasized the urgent need for transparent, robust behavioral analysis mechanisms [22].

To address these gaps, this thesis proposes a structured methodology for profiling the behavior of popular messaging applications on Android through kernel-level tracing using `ftrace` and `kprobes`. The proposed approach process behavioral patterns, aiming to enhance user privacy, and system transparency.

1.1 Motivation and Problem Statement

Motivation The motivation behind this research arises from the necessity to bridge existing gaps between user expectations, regulatory compliance, and the actual op-

erational behavior of popular messaging applications. Messaging apps process extensive personal data, creating substantial risks related to privacy violations and security breaches. Recent incidents involving unauthorized data collection by prominent messaging applications, along with revelations about governmental use of supposedly secure messaging platforms, underscore significant concerns regarding transparency and user trust.

Problem Statement Current literature lacks comprehensive kernel-level behavioral analyses of messaging applications, leaving critical privacy and security risks inadequately addressed. Thus, this research seeks to systematically explore kernel-level behaviors to enhance transparency, improve user trust, and provide rigorous technical evaluations of messaging applications' privacy implications.

1.2 Research Objectives

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

Primary Objectives

- Record the actual kernel-level behavior of widely used messaging applications.
- Identify potential violations of the principle of data minimization.
- Analyze mismatches between granted permissions and real-time resource usage.
- Detect unauthorized or hidden access to sensitive user data.
- Compare the behavioral profiles of privacy-focused apps (e.g., Signal) and more commercial alternatives.

Analytical and Technical Sub-Objectives

- 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 traced behaviors with declared permissions.
- Implement a web-based dashboard for behavior visualization.

Broader Goals

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

1.3 Research Questions

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

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

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

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

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

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

1.4 Limitations

- **Platform Scope**: Analysis restricted to Android 10+ due to kernel API dependencies.
- **Dynamic Analysis Constraints**: Real-world noise (e.g., background services) may affect system call traces.
- **App Selection Bias**: Focus on top-tier apps (WhatsApp, Signal, Telegram) may omit niche platforms.

1.5 Contributions of this Thesis

1.6 Thesis Outline

This thesis is organized into the following chapters:

- **Chapter 1 – Introduction:** Provides background context, outlines the motivation and objectives, presents the research questions, contributions, and a high-level overview of the thesis structure.
- **Chapter 2 – Related Work:** Reviews the scientific literature on Android application analysis with an emphasis on privacy-oriented behavioral profiling. Covers static and dynamic analysis techniques, system call modeling, and prior efforts on messaging app privacy. Synthesizes limitations of existing work and identifies the research gap this thesis addresses.
- **Chapter 3 – Technical Background:** Explains foundational Android OS mechanisms relevant to kernel-level tracing, including process lifecycle, Binder IPC, SELinux policies, the Android security model, and system call interfaces. Introduces messaging app architecture and privacy implications, contrasting static vs. dynamic analysis and justifying the need for kernel-level instrumentation.
- **Chapter 4 – Methodology and System Design:** Describes the research design, experimental setup, data collection using kernel-level tracing, and the analysis framework.
- **Chapter 5 – Results:** Presents the observed behavioral patterns, differences among messaging apps, and key findings related to privacy-relevant behaviors.
- **Chapter 6 – Discussion:** Interprets the results in light of the research questions, discusses limitations of the study, and suggests potential improvements.
- **Chapter 7 – Conclusions:** Summarizes key contributions, highlights findings, and suggests directions for future research.
- **Appendix A – Additional Data Tables:** Includes supplementary statistical data and traces.
- **Appendix B – Code:** Provides relevant shell scripts, Python tools, and configuration details used in the implementation.

Chapter 2

Related Work

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

2.1 Static Analysis of Android Applications

Static analysis inspects an *APK*'s bytecode off-line, allowing security vetting to run quickly, deterministically, and at scale. It therefore plays a foundational role in Android app auditing, offering early detection of potential vulnerabilities without requiring runtime execution. Over the last decade, four interwoven research strands have successively expanded the scope and robustness of static techniques.

(i) Privacy-oriented taint analysis. The first strand tracks information flows from sensitive *sources* (e.g. GPS, contacts) to security-critical *sinks* (e.g. network, logs). **FlowDroid** introduced context-, flow-, field- and object-sensitivity together with an accurate lifecycle model, reducing false alarms dramatically. On the *DroidBench* suite it reaches 93 % recall and 86 % precision—outperforming both academic and commercial tools [23]—and it scales to large real-world apps such as Facebook, PayPal and LinkedIn.

(ii) Inter-component communication (ICC). Many leaks traverse process boundaries. **IccTA** augments FlowDroid with an explicit ICC model, surpassing **Epicc** [25] and **CHEX** [26]—the latter targets component hijacking—on the *ICC-Bench* suite.

Amandroid extends the idea further by building a full inter-component data-flow graph linking intents, callbacks and content providers; on 753 Play-store apps and 100 malware samples it uncovered previously unknown OAuth-token and intent-injection flaws while keeping analysis under one minute per APK [27].

(iii) Obfuscation resilience. Reflection, dynamic loading and native libraries obscure control-flow and data types. **DroidRA** resolves reflective calls through constant-propagation and rewrites them into direct invocations, boosting FlowDroid's precision on reflection benchmarks by up to 60 % [29]. **DexLEGO** reassembles bytecode collected at run time [30], while **LibDroid** summarises taint propagation inside native libraries [33]; together they progressively widen the statically visible attack surface.

(iv) Learning-based malware detection. Moving beyond handcrafted rules, **Drebin** extracts lightweight syntactic features—permissions, intent actions, API calls—and detects malware with 94 % accuracy and 1 % false positives on 5 560 samples [28]. Deep networks now dominate: **DL-AMDet** combines CNN-BiLSTM embeddings with an auto-encoder to reach 99.9 % accuracy on public datasets [34]. Nonetheless, limited interpretability and training-set bias hinder deployment in high-assurance pipelines.

Evaluation benchmarks. *DroidBench* and *ICC-Bench* remain staples, but **TaintBench** adds complex real-world apps with documented ground truth [32]. The **µSE** framework mutates benign code to expose latent flaws, revealing thirteen previously undetected issues in a single toolchain and fuelling the debate on soundness versus scalability—dubbed "*soundiness*" by the community [31].

Remaining challenges. Implicit flows, callback re-entrancy, packed or virtualised code and the rapid churn of third-party SDKs still defeat static approximations. Current trends therefore favour *hybrid pipelines* that blend static precision with dynamic insight or combine symbolic reasoning with statistical heuristics.

2.2 User-Space Dynamic Instrumentation

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

(i) System-call monitors. **DroidTrace** hooks every `ptrace` event to log the full system-call stream and can *force* rare branches to execute, increasing coverage in malware analysis [35]. A later study learns n-gram models that reconstruct high-level Android API invocations purely from syscall sequences, demonstrating the semantic power of even low-level traces [36]. These tools deploy on stock ROMs and capture behaviour beneath any anti-hooking guard, yet they incur measurable per-call overhead and cannot observe Java-level reflection or IPC content.

(ii) Dynamic binary instrumentation (DBI). Instruction-granular DBI engines such as **Pin** (ported to ARM) enable fine-grained native tracing and custom analyses (e.g. memory-safety checks) with mature tooling support, but suffer from high runtime cost and outdated Android support. **QBDI** improves portability to modern ARM/Thumb-2 and withstands common packing techniques, although it remains native-only and requires root privileges [37].

(iii) In-process hooking frameworks. **Frida** injects a tiny loader that JIT-compiles JavaScript hooks into Java, JNI and native code, permitting live patching and interactive experiments [39]. **DYNAMO** layers systematic diff-based analysis on top of Frida to study framework evolution across Android versions [38], while **InviSeal** hardens Frida against detection, shrinking overhead to $< 3\%$ on UI benchmarks [40]. Yet all three struggle with tamper-aware apps and expose only the instrumented process—cross-app flows, scheduler activity and kernel events remain hidden.

(iv) Sandbox and multi-layer profilers. Containerised sandboxes such as **C-Android** isolate each app inside a Linux namespace and stream user-space logs for reproducible testing [41]. Emulation-centred pipelines (e.g. **DynaLog** or DroidBox derivatives) automate large-scale malware triage by collecting $\sim$100 runtime features in parallel VMs [42]. Although high-throughput, these environments diverge from real-device timing, miss in-kernel I/O, and are easily fingerprinted by sophisticated apps.

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

2.3 Kernel-Level Tracing on Android

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

(i) eBPF- and ftrace-based syscall/Binder collectors. **BPFroid** streams per-UID syscalls, Binder calls and network events from *stock* Android kernels to a real-time malware detector, achieving $\sim$3 % overhead and 70 % Genome detection accuracy at run time [43]. Earlier efforts, such as **ID-Syscall**, compiled a loadable kernel module to log syscall sequences for anomaly detection on Android 4.x [44]. These works show that rich behavioural signals are available without modifying the firmware image, yet they focus on binary classification rather than fine-grained messaging-app semantics.

(ii) Network-centric and covert-channel tracing. Using vanilla *ftrace*, Celik and Gligor detect *stegomalware* by spotting timing and packet-size irregularities in kernel-level traces [45]. Their methodology—correlating scheduler, socket and power events—is transferable to instant-messaging clients, but the study evaluates only artificial channels and omits Binder traffic.

(iii) Hardware-assisted monitors. **HART** leverages ARM Embedded Trace Macrocell (ETM) to stream control-flow of binary-only kernel modules with zero software overhead [46]. While ideal for diagnosing vendor drivers that gate camera or GPU access in chat apps, ETM is often disabled on production handsets and reveals no user-process context.

(iv) Production-scale, low-overhead tracing. Google's **Perfetto** combines *ftrace*, Binder and userspace counters into multi-gigabyte traces that can be queried in SQL to dissect app start-up and jank [47]. Complementarily, **eMook** shows how per-PID filters inside eBPF eliminate tracing overhead on unrelated processes, cutting system cost by 85 % [48]. These advances make week-long field studies of battery-powered phones feasible but have yet to target security or messaging use-cases explicitly.

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

2.4 Privacy Studies on Messaging Applications

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

(i) On-device artefacts and forensics. Early work focused on residual data stored locally. Anglano's systematic analysis of WhatsApp on Android 4.x recovered chat databases, media thumbnails and contact lists even after user-initiated deletions [50]. 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 [51, 52]. While modern versions encrypt local backups, notification side-channels (e.g. Firebase tokens) remain an open avenue [53].

(ii) Network-level traffic analysis. A second strand exploits packet timing, size and sequence to infer user actions without breaching encryption. Pioneering work by Matic *et al.* fingerprinted iMessage events (send, read, typing) with > 90 % accuracy solely from TLS record bursts [54]. Apthorpe demonstrated similar leakage for Android IM apps over Wi-Fi [55]. Recent machine-learning classifiers identify not only event types but also multimedia uploads and voice-over-IP handshakes in QUIC traffic [56]. Countermeasures such as adaptive padding raise bandwidth by 8–14 % while mitigating most classifiers [57].

(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 [58]. Kwon showed that Telegram's "People Nearby" feature enables trilateration of user location with meter-level accuracy [59].

Signal introduced private-set-intersection with SGX enclaves to curb this vector, yet follow-up work revealed side-channels via cache timing [60].

(iv) Metadata policies and compliance audits. A complementary line audits whether apps meet their self-declared privacy policies. Egele's longitudinal crawl of Telegram channels uncovered the retention of deleted media on CDN edge nodes weeks after deletion [61]. Bock dissected Signal's sealed-sender protocol, confirming that only sender and receiver persist in a 24-hour service log but flagging exposure of push-token identifiers to Apple/Google [62]. Recent GDPR-oriented studies show that less than 40 % of messaging apps provide complete data-export facilities despite legal requirements [63].

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

Other Complementary Approaches

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

Grey-box fuzzing. Coverage-guided fuzzers such as *AFL++* and cloud services like *ClusterFuzz* generate mutated inputs that drive apps along rarely executed paths. Although they do not rely on full program knowledge, they systematically surface crashes and logic errors that can later be inspected with dynamic or kernel monitors [64, 65].

Formal verification and model checking. Specification languages (e.g., TLA$^+$) combined with deductive verifiers (e.g., VeriFast) enable proofs of safety properties—such as correct handshake sequences in secure-messaging protocols or the absence of data races in JNI bridges—without collecting runtime traces [66, 67].

Privacy-preserving telemetry. Large-scale differential-privacy frameworks adopted by Apple and Google collect aggregate statistics while bounding individual leakage, offering population-level insight into security events without intruding on single devices [68, 69].

Crowdsourced UI exploration. Robo-testing engines (e.g., Firebase Test Lab) drive accessibility events to map interface flows that static call graphs miss, producing semantic traces useful for regression testing and security auditing [70].

Physical side-channel analysis. Power-monitoring boards and electromagnetic probes can infer sensitive activity—such as encryption or radio bursts—without any software hooks, complementing software-based tracing when root or debugging privileges are unavailable [71].

Human-subject studies. Qualitative interviews and ethnographic methods shed light on adoption hurdles, configuration practices, and privacy norms that remain invisible to purely technical traces [72].

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

2.5 Synthesis and Research Gap

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

Synthesis of Current Methodologies

- **Static analysis** offers breadth and determinism, uncovering dormant code paths and permission misalignments at scale. Its reliability, however, is bounded by obfuscation, dynamic loading, and absent runtime context.
- **User-space instrumentation** provides rich semantic detail—method names, parameters, UI state—and deploys quickly on stock devices. This power is tem-

pered by visibility gaps beneath the syscall layer, susceptibility to anti-hooking defences, and non-negligible overhead.

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

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

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

Identified Research Gap

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

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

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

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

Chapter 3

Technical Background

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

3.1 Android Architecture and Kernel-Level Access

3.1.1 Android Software Stack Overview

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

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

Beneath the framework lies the Android Runtime (ART), which executes appli-

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

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

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

Figure 1: Updated Diagram of Android Software Stack (source: Android Developers Guide [**?**]).

3.1.2 Application Layer and Process Lifecycle

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

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

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

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

Understanding transitions between app states (e.g., from idle to foreground) is vital in syscall-level profiling. For instance, foreground activation often leads to bursts of system calls such as `open()`, `stat()`, and `mmap()`—associated with UI initialization and resource loading. Such behaviors form recognizable patterns in kernel trace logs. **Figure 2:** Android Application Lifecycle and Corresponding Kernel-Level Events.

3.1.3 Android Runtime, Native Layer, and JNI

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

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

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

In the context of this thesis, which focuses on dynamic kernel-level analysis, capturing system interactions initiated by ART and JNI is essential. It enables the identification of execution phases or modules that deviate from expected behavior—especially in apps that rely heavily on native components for messaging, encryption, or background communication. **Figure 3:** JNI and ART Interaction with Kernel during JIT and Native Execution.

3.1.4 Linux Kernel Fundamentals in Android

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

system resources are mediated through kernel functions and system calls.

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

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

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

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

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

3.1.5 System Calls and Kernel Interaction

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

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

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

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

3.1.6 Android Security Model and Isolation Mechanisms

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

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

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

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

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

3.2 Messaging Apps: Characteristics Privacy Implications

3.2.1 Functional and Architectural Overview

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

Android, as the dominant mobile operating system, provides the primary distribution platform for these apps through the Google Play Store. According to public data [**?**, **?**], Facebook Messenger has surpassed 5 billion downloads, Telegram exceeds 1.2

billion downloads, and Signal has more than 100 million installs. While usage varies by region, these numbers highlight the ubiquity and market penetration of messaging applications on Android devices.

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

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

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

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

3.2.2 Privacy-Critical Behaviors and Resource Usage

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

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

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

unrelated to core messaging functionality [**?**].

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

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

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

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

3.2.3 Architectures, Privacy, and Cryptographic Models

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

Signal utilizes a centralized yet privacy-focused architecture, relying exclusively on the Signal Protocol, a robust end-to-end encryption (E2EE) scheme ensuring forward secrecy, deniability, session-specific ephemeral keys, and the Double Ratchet algorithm for key management. The Double Ratchet combines a Diffie-Hellman key exchange and symmetric key cryptography, generating new encryption keys for every message sent, significantly enhancing security against key compromise [49]. 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 syn-

chronization and external dependencies, significantly reducing its syscall footprint.

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

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

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

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

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

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

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

3.3 Static vs. Dynamic Analysis

The security and behavioral analysis of Android applications can be approached through two primary methodologies: static and dynamic analysis. Each offers unique advantages and limitations, particularly when it comes to uncovering privacy-sensitive behavior in messaging applications. This section contrasts the two approaches and motivates the use of kernel-level dynamic analysis as the foundation for the profiling methodology employed in this thesis.

3.3.1 Static Analysis

Static analysis examines the application code, configuration files, and resources without executing the application. Tools such as JADX, Androguard, and MobSF decompile APKs and provide insights into declared permissions, control flow structures, and third-party library usage. This method is effective for identifying potentially dangerous code paths, hardcoded secrets, or violations of best practices.

However, static analysis suffers from several well-documented limitations. It cannot reliably account for runtime behavior that is conditional, obfuscated, or implemented in dynamically loaded or native code. Additionally, it provides no visibility into real-world execution patterns, user interactions, or background operations initiated through APIs like `JobScheduler` or `AlarmManager`. From a privacy standpoint, static analysis cannot reveal whether declared permissions are exercised legitimately or abused in practice.

3.3.2 Dynamic Analysis and Kernel-Level Tracing

Dynamic analysis captures application behavior during execution, providing ground-truth data on runtime interactions with the operating system. Techniques range from high-level instrumentation using frameworks like Frida and Xposed, to lower-level monitoring using strace or custom logging APIs. For this thesis, we adopt kernel-level dynamic analysis, specifically through system call tracing using tools such as `ftrace`, `kprobes`, and `debugfs`.

Kernel-level tracing offers a privileged vantage point for observing all user-to-kernel transitions, regardless of the programming language, framework, or runtime environment used by the application. Unlike instrumentation at the application layer, which can be bypassed by obfuscation or native code execution, kernel tracing captures raw syscall activity including file I/O (`open()`, `read()`, `unlink()`), network operations (`connect()`, `send()`, `recvmsg()`), and cryptographic processing (`getrandom()`, `mprotect()`, `write()`).

This approach is especially relevant for profiling messaging applications, which often perform background operations, access sensitive resources, and engage in encrypted communications without user interaction. By analyzing syscall sequences and their temporal patterns, one can construct behavioral profiles that are resilient to static evasion techniques and indicative of privacy-relevant behavior.

Figure 8: Comparison of Static Analysis, Dynamic Instrumentation, and Kernel-Level Tracing Capabilities.

3.4 System Call Tracing and Behavioral Analysis

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

3.4.1 Tracing Interfaces and Instrumentation Tools

Several kernel-level tracing tools are available in Android and Linux-based systems, each offering a distinct trade-off between overhead, observability, and programmability:

- **ftrace**: Integrated into the Linux kernel, ftrace provides function-level and syscall-level tracing, context switches, scheduling events, and filtering by PID or syscall. It is accessible via `/sys/kernel/debug/tracing` and optimized for minimal overhead.
- **kprobes and uprobes**: Support dynamic, on-the-fly instrumentation at both kernel and user-space function addresses. Useful for attaching probes at points like `do_sys_open`, logging file accesses and argument values.
- **tracepoints**: Provide predefined hook points in the kernel source that offer safe, efficient monitoring of events such as process creation, file access, or scheduler activity. Often used with tools like perf or BCC.
- **strace**: A user-space syscall tracer that uses ptrace to log syscall invocations. Ideal for debugging but with high overhead and poor stealth, making it unsuitable for continuous profiling.
- **eBPF (Extended Berkeley Packet Filter)**: A modern programmable tracing interface allowing safe kernel extensions at runtime. It supports dynamic filtering, per-event logic, aggregation, and export to user space. Its adoption in Android is growing with newer kernels.

Table 1: Comparison of Kernel Tracing Tools

3.4.2 Raw Trace Data and Post-Processing

The output from syscall tracing tools typically consists of structured logs, such as:

```
<...>-1234 [000] .... 10000.123456: sys_enter_openat: dfd=AT_FDCWD filename="/data/
```

These logs capture syscall names, parameters, timestamps, and context (e.g., PID, thread state). To extract insights:

- Data is filtered and cleaned to isolate app-specific syscall events.
- Features such as syscall frequency, argument patterns, and return codes are extracted.
- Aggregated traces are converted to higher-level representations (e.g., n-grams, histograms) for machine learning or statistical analysis.

Post-processing typically involves parsing tools like Python scripts, custom Bash pipelines, or tools like `perf`, `bcc`, and `Sysdig`. These allow for correlation with app state (foreground/background), event type, and time windows.

2.4.3 System Call Pattern Analysis and Behavioral Fingerprinting

Figure 9: System Call Tracing Pipeline and Example Behavioral Fingerprints for Messaging Apps.

Chapter 4

Methodology and System Design

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

4.1 Research Design

The research design employs a kernel-level tracing approach to profile behavioral patterns of popular Android messaging applications without requiring app instrumentation. The primary goal is to transparently capture system-level activities, such as system calls, IPC, and network interactions, to detect privacy-sensitive behaviors. The central research question is: *"How effectively can kernel-level tracing detect privacy-sensitive behaviors in messaging apps?"*

Kernel-level tracing was chosen over user-level or API-based approaches due to its transparency, accuracy, and non-invasive nature. Experiments were performed on a rooted Android device (OnePlus Nord CE 4 Lite, Android 15) using Magisk and bootloader unlocking for kernel tracing filesystem (tracefs) access.

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

An additional design decision was the use of kprobes and ftrace instead of more advanced or emerging technologies such as eBPF. This choice was primarily driven by the principle of simplicity, in line with Occam's razor—opting for the simplest effective

solution when more complex alternatives are unnecessary. Since the research objectives could be fully met using established, low-level tracing tools, there was no compelling need to adopt or upgrade to more sophisticated frameworks. This approach facilitated ease of deployment, lower learning curve, and minimized potential system instability or compatibility issues, while still ensuring comprehensive data collection for the study's requirements.

Design principles such as reproducibility, low overhead, and the detection capability for IPC, sensitive resource access, and network activities are prioritized. A sliding window approach (5000 events per window, 1000-event overlap) is applied for detailed temporal analysis.

Data quality is ensured via validation and heuristic cleaning to filter irrelevant daemon activities. Special attention is given to privacy protection, ensuring sensitive resource monitoring without data compromise. Comprehensive downstream analysis, including statistical evaluation, visualization, and a web-based interface, will be thoroughly detailed in subsequent sections.

4.2 Experimental Setup

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

Initially, the device was prepared for root access through 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.

Using PowerShell, the device was confirmed to be recognized via the command `adb devices`, followed by booting into bootloader mode using `adb reboot bootloader`. The bootloader was unlocked with `fastboot flashing unlock`, confirmed on the device's screen, and rebooted.

To obtain the boot image, the full OTA zip for the CPH2621 EU variant was downloaded via Oxygen Updater, transferred to the computer using `adb pull`, and extracted to obtain the `payload.bin`. The Payload Dumper tool was utilized (`python payload_dumper.py payload.bin`) to extract the `boot.img`.

Subsequently, Magisk APK was installed, the boot image was patched, and the patched image was flashed using `fastboot flash boot_a magisk_patched-28100_4owcs.img`.

Despite initial issues, careful attention to documentation led to the correct flash method using the `init_boot.img`.

The ADB setup enabled detailed interaction between the computer and the rooted device, allowing the execution of custom shell scripts for tracing (e.g., `simple_trace_sock.sh`) 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 comprehensive and precise experimental configuration provided the basis for reliable kernel-level data collection, ensuring reproducibility, accuracy, and minimal system intrusion.

In order to associate kernel-level trace events with high-level resources (e.g., camera, microphone, contacts database), it was necessary to extract device identifiers from the target Android device. This process involved executing shell commands via ADB to map each device class to its corresponding major:minor identifier. Specifically, a custom loop over the /sys/class/**/dev entries was used to enumerate character and block devices, using the formula `dev_t = (major << 20) | minor`. This allowed us to collect and encode device IDs such as video0, nq-nci, btpower, pcmC0D0c, and GNSS/NFC interfaces.

Additionally, to monitor access to sensitive databases (e.g., contacts2.db, mmssms.db), file paths were manually located, and their inode values were extracted using stat commands. Each resource was annotated with metadata including its full path, description, major, minor, and inode values. These identifiers enabled accurate matching with trace events during analysis.

To streamline and automate this otherwise manual and error-prone process, a collaborator (Giannis Karyotakis) contributed custom Bash and Python scripts that programmatically retrieved and serialized this metadata into JSON files. These mappings were then used during the parsing phase to label events associated with sensitive resources.

Identifying these device and file descriptors was essential for enabling high-level semantic interpretation of low-level events, supporting accurate privacy-relevant event detection throughout the tracing pipeline.

This comprehensive and precise experimental configuration provided the basis for reliable kernel-level data collection, ensuring reproducibility, accuracy, and minimal system intrusion.

4.3 Data Collection and Preparation

Kernel tracing involved the detailed instrumentation of kernel events, specifically system calls associated with file operations, IPC mechanisms, network activities, and device resource access. Data collection scripts configured kernel trace buffers and event triggers, executing real-time logging with precise timestamping and process identifiers (PIDs/TGIDs).

Data preparation comprised multiple stages:

1. **Raw Data Acquisition**: Collection of kernel trace logs directly from the device using custom shell scripts.
2. **Data Extraction and Parsing**: Python-based utilities parsed raw ftrace logs into structured JSON formats, extracting event types, timestamps, arguments, and associated process metadata.
3. **Filtering and Cleaning**: Heuristic-driven preprocessing methods removed irrelevant data, background noise (e.g., system daemon activities), and redundant event entries, yielding refined event sequences.
4. **Event Slicing**: Implemented sliding window techniques with user-defined overlap (5000 events per window, 1000 events overlap) to enable detailed temporal analysis of app behaviors.

4.4 Feature Extraction from System Calls

4.5 System Architecture and Data Flow

The implementation is organised as a series of modular layers that cooperate to capture, process, analyse, and visualise kernel-level trace events. Figure 4.1 summarises the architecture and the principal data-flow.

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

4.5.1 Codebase Structure

The repository is divided into coherent top-level modules:

`config_files/` Declarative lists of tracepoints and kprobe hooks; filters for processes, devices, and events.

`scripts/` On-device shell utilities: `simple_trace_sock.sh` (start/stop tracing), `find_rdev_stdev_ir` (enumerate major:minor and inode mappings), and helper Python wrappers.

`data/traces/` Raw `*.trace` logs collected from the device; `data/mappings/` stores JSON maps of logical resource categories (*camera*, *bluetooth*, *contacts* ...).

`app/` Core analysis library (`myutils.py`), slicing engine, and Jupyter notebooks for exploratory study.

`Exports/` Cleansed and sliced outputs (CSV — JSON) consumed by the visual dashboard or external statistics packages.

`webapp/` Flask backend plus D3.js/Bootstrap front-end that renders interactive timelines, pie charts, and heat-maps.

4.5.2 Instrumentation Layer

Tracing scripts load `kprobes` via `tracefs` to intercept syscall entry/exit points (`read_probe`, `write_probe`, `ioctl_probe`) and network or IPC events (`inet_sock_set_state`, Binder transaction tracepoints). Configuration files permit selective activation per device, process, or resource, while nanosecond timestamps preserve causal ordering.

4.5.3 Parsing and Pre-processing

A two-stage parser converts raw ftrace lines to structured Python dictionaries. A primary regex extracts static fields; a secondary pass tokenises `key=value` pairs and casts numeric values (decimal or hexadecimal). Three graded cleaners remove noise—system daemons, ephemeral file descriptors, spurious Binder ioctl cycles—retaining only semantically meaningful user-level events.

4.5.4 Heuristic Slicing Engine

The slicing engine applies forward and backward analyses around each API invocation window.

- *Forward slice*: follows writes, ioctls, and socket state transitions originating from the API thread, recursively including child processes contacted via Binder or UNIX sockets.
- *Backward slice*: traces reads and ioctls that supply data to the same window.

Causal graphs are reconstructed implicitly by maintaining dynamic PID sets and Binder transaction IDs. Complexity is $\mathcal{O}(N, M)$ per window (N events, M instances) and is mitigated by aggressive pre-filtering.

4.5.5 Export and Visual Analytics

Sliced sequences are serialised as compact JSON maps (`api_code → [in,out]`) and tabular CSV. A Flask API exposes timeline and summary statistics, while a D3.js front-end offers interactive filtering by PID, device ID, or time range.

4.5.6 End-to-End Flow

`config_files/ + scripts/` → raw trace via `tracefs` → parsing → cleaning/indexing → slicing → JSON/CSV export → Flask API → D3.js visualisation

This modular pipeline facilitates future extensions—e.g., substituting a graph-database backend or integrating user-space probes—without perturbing the upstream instrumentation layer.

4.6 Data Visualization and User Interface Design

To facilitate high-level understanding of kernel-level behaviors, the visualization strategy incorporates:

- **Interactive Event Timelines**: Provide detailed temporal views of categorized system activities.
- **Statistical Summaries and Graphs**: Depict device usage patterns, frequency of access, and event-type distributions using charts (pie, bar, heatmaps).
- **Dynamic Filtering and Interaction**: Allow users to filter visualizations dynamically based on PID, event types, and accessed devices.

Chapter 5

Results

5.1 Behavioral Patterns Observed

5.2 Model Performance

5.3 Comparison Between Messaging Apps

5.4 Classification or Pattern Recognition Outcomes

Presentation of evaluation tables, charts, and analysis derived from the ML algorithms.

5.5 Comparisons and Interpretations

Comparison of different models or configurations, with emphasis on interpreting discrepancies and assessing each model's performance.

5.6 Visualizations

Chapter 6

Discussion

6.1 Interpretation of Results

6.2 Limitations of the Study

6.3 Opportunities for Improvement

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

Chapter 7

Conclusions

7.1 Key Findings

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

7.2 Future Research Directions

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

Bibliography

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

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

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

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

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

[6] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[23] Arzt C., Rasthofer S., Bodden E. *et al.* FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. *Proceedings of PLDI*, 2014. bodden.de

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[59] Kwon D. Triangulating Telegram Users via "People Nearby". *S&P*, 2021.

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

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

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

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

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

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

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

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

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

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

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

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

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

Appendix A

Appendix A: Additional Data Tables

Any further data tables, graphics, or supplementary material.

Appendix B

Appendix B: Code

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