Athens University of Economics and Business

Department of Management Science and Technology

Undergraduate Thesis

Behavioral Profiling of Popular Messaging Apps Using Kernel-Level Tracing with ML Techniques

Student:

Foivos - Timotheos Proestakis

Student ID: 8210126

Supervisors:

Prof. Diomidis Spinellis

Dr. Nikolaos Alexopoulos

Submission Date:

April 5, 2025

Abstract

This thesis examines kernel-level tracing techniques to create behavioral profiles of popular messaging applications using Machine Learning. The main goal is to analyze the operational characteristics of such apps and employ ML algorithms to detect patterns regarding security. The study covers topics such as kernel-level data collection, big data processing and analysis, and the design of ML models for behavior identification and classification.

Acknowledgments

I would like to express my heartfelt 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 [10, 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 integrates Machine Learning techniques to process and classify behavioral patterns, aiming to enhance security diagnostics, user privacy, and system transparency.

1.1 Motivation and Problem Statement

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

Problem Statement 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.
- Identify potential indirect data leakage through side-channel patterns.
- 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 and Technical Background:** Reviews existing literature on Android architecture, messaging app privacy implications, static and dynamic analysis techniques, system call tracing, and identifies key research gaps.
- **Chapter 3 – Methodology and System Design:** Describes the research design, experimental setup, data collection using kernel-level tracing, and the analysis framework.
- **Chapter 4 – Results:** Presents the observed behavioral patterns, differences among messaging apps, and key findings related to privacy-relevant behaviors.
- **Chapter 5 – Discussion:** Interprets the results in light of the research questions, discusses limitations of the study, and suggests potential improvements.
- **Chapter 6 – 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 and Technical Background

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.

This chapter provides a detailed overview of related work and 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, and identifies gaps in existing research where this thesis contributes.

2.1 Android Architecture and Kernel-Level Access

2.1.1 Android Software Stack Overview

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

the Java API Framework (commonly referred to as the Application Framework), the Hardware Abstraction Layer (HAL), and the Linux Kernel.

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

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

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

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

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

2.1.2 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.

2.1.3 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.

2.1.4 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.

2.1.5 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.

2.1.6 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.

Figures and Diagrams

- Figure 1: Android Software Stack
- Figure 2: Application Lifecycle
- Figure 3: Kernel and User-Space Separation
- Figure 4: Binder IPC Mechanism

- Figure 5: SELinux and Access Control Flow
- Figure 6: System Call Interaction Flow
- Figure 7: Kernel Tracing Pipeline

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

2.2 Messaging Apps: Characteristics Privacy Implications

2.3 Static vs. Dynamic Analysis

2.4 System Call Analysis and Kernel Tracing

2.5 Machine Learning for Behavior Profiling

2.6 Related Research Gaps in Literature

Chapter 3

Methodology and System Design

3.1 Research Design

3.2 Experimental Setup

A discussion of experimental settings, including how experiments were conducted and what evaluation metrics (e.g., accuracy, precision, recall, F1-score) were used.

3.3 Data Collection and Preparation

A detailed description of how kernel-level data is collected and the preprocessing steps taken to ensure suitability for ML algorithms.

3.4 Feature Extraction from System Calls

3.5 Machine Learning Models and Tools

An overview of the ML algorithms (e.g., Random Forest, SVM, Neural Networks) and the software tools (e.g., Python, scikit-learn) employed in the study.

Chapter 4

Results

4.1 Behavioral Patterns Observed

4.2 Model Performance

4.3 Comparison Between Messaging Apps

4.4 Classification or Pattern Recognition Outcomes

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

4.5 Comparisons and Interpretations

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

4.6 Visualizations

Chapter 5

Discussion

5.1 Interpretation of Results

5.2 Limitations of the Study

5.3 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 6

Conclusions

6.1 Key Findings

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

6.2 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.

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.