TU WIEN Informatics

Virtualization-based Code Obfuscation via Android Runtime Permutation

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Mario Lecker, BSc
Matrikelnummer 01634035

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing.in Dr.in techn. Martina Lindorfer, BSc
Mitwirkung: Jakob Bleier, BSc BSc MSc

Wien, 12. Februar 2026

Mario Lecker

Associate Prof. Dipl.-Ing.in Dr.in techn. Martina Lindorfer, BSc

TU WIEN Informatics

Virtualization-based Code Obfuscation via Android Runtime Permutation

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Mario Lecker, BSc
Registration Number 01634035

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Ing.in Dr.in techn. Martina Lindorfer, BSc
Assistance: Jakob Bleier, BSc BSc MSc

Vienna, February 12, 2026

Mario Lecker

Associate Prof. Dipl.-Ing.in Dr.in techn. Martina Lindorfer, BSc

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Mario Lecker, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, habe ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT-Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 12. Februar 2026

Mario Lecker

Acknowledgements

First and foremost, my sincere gratitude goes to my supervisors, Prof. Martina Lindorfer and Jakob Bleier. I am deeply thankful that you gave me the chance to pursue this research when I approached you with an existing master thesis topic. Your belief in me and your willingness to support my vision were fundamental to the success of this work.

The unwavering support of my family has also been invaluable. To my father, thank you for nurturing my technical interests and my love for building things from a young age. Those early years spent constructing and deconstructing with colorful plastic bricks laid the foundation for the engineer I am today.

I owe a special debt of gratitude to my mother. She has always provided an open ear for my struggles, listening with endless patience, even when I needed to vent at length about the difficulties I faced. Her comforting words were a constant source of strength that kept me grounded and helped me push through the toughest moments.

Finally, my deepest love and gratitude belong to my significant other. Thank you for standing by my side and for being so understanding throughout this challenging journey. Your encouragement to stay focused was a constant source of motivation, and simply having you in my life makes every challenge easier to face.

Kurzfassung

Mobile Anwendungen erfordern einen robusten Schutz, um die Kosten für Reverse Engineering zu maximieren und die Wahrscheinlichkeit erfolgreicher Angriffe zu verringern. Traditionelle Verschleierungsmethoden wie Identifier Renaming ermöglichen weiterhin eine statische Analyse der Bytecode-Struktur. Die Virtualisierung auf Anwendungsebene verschleiert den Kontrollfluss zwar effektiv, beeinträchtigt jedoch die Leistung aufgrund häufiger Kontextwechsel über das Java Native Interface (JNI). Wir schlagen daher ein Virtualisierungsframework auf Systemebene vor, das Instruction Set Randomization (ISR) direkt in die Android Runtime (ART) integriert.

Wir stellen das Android Opcode Permutation Tool (AOPT) vor, mit dem Standard-Dalvik-Bytecodes von Android-Anwendungen in einen randomisierten Befehlssatz transformiert werden. Das Tool permutiert Opcode-Werte bijektiv, fügt ein Metadaten-Flag in das Android-Manifest ein und generiert eine Opcode-Zuordnung, die als symmetrischer Schlüssel dient. Um die Ausführung dieser permutierten Anwendungen zu ermöglichen, erstellen wir eine angepasste Version des Android 15-Betriebssystems. Das Betriebssystem erkennt das Metadaten-Flag und leitet es an die ART weiter. Wir gewährleisten Abwärtskompatibilität, indem wir den Fallback-Switch-Interpreter erweitern und mehrere Befehlssatzarchitekturen unterstützen. Der Interpreter verwendet die Opcode-Zuordnung für die dynamische Suche, um die gleichzeitige Ausführung von Standard- und verschleierten Anwendungen zu ermöglichen.

Wir evaluieren die Robustheit des Frameworks anhand von 20 realen Anwendungen. Der Testkorpus umfasst komplexe Software wie z.B. Webbrowser, Medieneditoren und kryptografische Finanztools. Die Verschleierungstechnik erreicht eine Kompatibilitätsrate von 85% bei vernachlässigbaren Auswirkungen auf die Startlatenz. Algorithmische Benchmarks zeigen einen erhöhten Ausführungsaufwand zwischen 11% und 30% aufgrund der Opcode-Suche. Dieser Kompromiss bei der Leistung führt zu einer erheblichen Widerstandsfähigkeit gegen Reverse Engineering. Standard-Decompiler wie JADX generieren semantisch falschen Code. Ebenso können dynamische Instrumentierungstools wie Frida keine Methoden hooken, da sie auf Standard-Laufzeitstrukturen angewiesen sind. Folglich zielt das Framework auf Hochsicherheitsumgebungen ab, in denen die organisatorische Firmware-Kontrolle und der robuste Schutz den Verlust der Portabilität rechtfertigen.

Schlüsselwörter: Softwareverschleierung, Android Runtime, Opcode Permutation, Mobile Sicherheit, Reverse Engineering, Virtualisierungsbasierte Verschleierung

Abstract

Mobile applications require robust protection to maximize the cost of reverse engineering and reduce the likelihood of successful attacks. Traditional obfuscation techniques, such as identifier renaming, leave the bytecode structure exposed to static analysis. Application-level virtualization effectively obfuscates control flow but degrades performance through frequent context switching via the Java Native Interface (JNI). Therefore, this thesis proposes a system-level virtualization framework that integrates Instruction Set Randomization (ISR) directly into the Android Runtime (ART).

We introduce the Android Opcode Permutation Tool (AOPT) to transform Android applications by converting standard Dalvik bytecode into a randomized instruction set. The tool permutes opcode values bijectively, injects a metadata flag into the Android manifest, and generates an opcode mapping that serves as a symmetric key. To enable the execution of these permuted applications, we create a custom version of the Android 15 operating system. The operating system detects the metadata flag and propagates it to the ART. We ensure backwards compatibility by extending the fallback switch interpreter to support multiple instruction set architectures. The interpreter uses the opcode mapping for dynamic lookup to enable the simultaneous execution of standard and obfuscated applications.

We validate the framework's robustness using a suite of 20 complex, real-world applications. The test corpus includes computationally demanding software such as full-featured web browsers, media editors, and cryptographic financial tools. The solution achieves an 85% compatibility rate with negligible impact on launch latency. Algorithmic benchmarks reveal an execution overhead between 11% and 30% due to mandatory opcode lookup. However, this performance trade-off yields substantial resilience against reverse engineering. Standard decompilers such as JADX generate semantically incorrect code. Similarly, dynamic instrumentation tools such as Frida fail to hook methods because they rely on standard runtime structures. Consequently, the framework targets high-security environments where the organization retains firmware control. In such contexts, the robust protection justifies the loss of portability.

Keywords: software obfuscation, Android Runtime, opcode permutation, mobile security, reverse engineering, virtualization-based obfuscation

Contents

CHAPTER 1

Introduction

1.1 Motivation

Mobile computing has become an integral part of modern society due to the exponential growth of mobile applications and the increasing reliance on smartphones for various personal and professional tasks. Android holds approximately 71% of the global mobile operating system market share and powers billions of devices [53]. As a result, both benign and malicious actors seek to protect their mobile applications from attacks. They aim to maximize the cost of reverse engineering until it outweighs the potential gain.

Standard Android build tools, such as ProGuard [54] and R8 [55], apply simple identifier renaming. This technique leaves the underlying Dalvik bytecode structure and framework API calls exposed to static analysis. Advanced obfuscation techniques such as junk code insertion and control flow flattening increase the structural complexity of the application [56]. However, automated deobfuscators and symbolic execution tools frequently reverse these modifications. Encryption also fails to provide sufficient security. It protects code only at rest, as the runtime decrypts the payload into memory before execution. Code virtualization addresses this limitation by decoupling the program code from the standard execution environment.

Traditional virtualization-based obfuscation techniques typically embed a custom interpreter within the application's native libraries [57]. This architecture bottlenecks performance due to frequent context switching between the managed runtime and the native environment via the Java Native Interface (JNI). Furthermore, the interpreter logic resides in a static native library and remains susceptible to binary analysis tools. Developers mitigate these performance penalties by obfuscating only individual code sections. This manual selection burdens developers and sacrifices the security of the overall application.

Therefore, we require a novel solution that enhances application security while preserving efficiency and responsiveness. Modifying the Android Runtime (ART) directly offers a high-performing alternative to designing a new virtual machine. This system-level approach eliminates the JNI bottleneck and integrates obfuscation logic into the core execution engine. It enforces a unique, randomized instruction set at the operating system level and protects application logic from standard reverse engineering tools.

1.2 Research Questions

The following research questions (RQs) serve as the primary goals of this thesis:

- **RQ1:** What are the most common traditional code obfuscation techniques, and how can these insights be leveraged to design advanced obfuscation techniques within the Android ecosystem?
- **RQ2**: How can the Android Runtime be modified to support a state-of-the-art virtualization-based obfuscation technique, and what requirements are necessary for a successful implementation?
- **RQ3**: How effective is the implemented virtualization-based obfuscation technique in fortifying the security of Android applications against various methods of reverse engineering, and what quality criteria are affected by applying the obfuscation?

1.3 Contributions

This thesis presents a novel system-level virtualization framework for the Android ecosystem. We make the following primary contributions:

- **Bijective Opcode Permutation Algorithm:** We design an Instruction Set Randomization (ISR) algorithm that transforms the standard Dalvik instruction set into a unique, randomized instruction set. This algorithm creates a one-to-one mapping for valid opcodes and ensures that each application utilizes a distinct instruction set architecture.
- **Android Opcode Permutation Tool (AOPT):** We develop a Java-based toolchain to automate the obfuscation and repackaging process. This tool reads arbitrary APK files, applies the permutation algorithm to the bytecode, and injects a metadata flag into the Android manifest.
- **Custom Android Runtime Interpreter:** We modify the Android Runtime within the Android Open Source Project (AOSP) to execute permuted bytecode. The runtime detects the obfuscation flag and dynamically resolves randomized opcodes. This proof-of-concept implementation extends the standard C++ switch interpreter.

1.4 Structure of Thesis

The remainder of this thesis is structured as follows: Chapter 2 provides the necessary background on the Android platform architecture, its security model, and common mobile reverse engineering techniques. Chapter 3 reviews related work in the field of static analysis and existing code obfuscation research. Chapter 4 analyzes traditional obfuscation approaches and discusses their limitations in protecting Android applications. Chapter 5 presents the core contribution of this work, detailing the design and implementation of the AOPT toolchain and the modified Android Runtime. Chapter 6 evaluates the proposed solution regarding functional compatibility, performance overhead, and security resilience against reverse engineering. Finally, Chapter 7 summarizes the findings and outlines potential directions for future research.

CHAPTER 2

Background

This chapter covers fundamental background knowledge necessary for understanding the proposed virtualization-based obfuscation mechanism. Section 2.1 analyzes the Android system architecture and the execution environment of the Android Runtime. Section 2.2 details the structure of Android applications and the compilation process of Dalvik bytecode. Section 2.3 outlines the Android security model and its core protection mechanisms. Finally, Section 2.4 categorizes static and dynamic reverse engineering techniques that threaten application confidentiality.

2.1 Android

Android is an operating system for mobile devices and is maintained by Google and the Open Handset Alliance [58, 59]. At the time of writing, Android holds approximately 71% of the global mobile operating system market share [53]. Xun et al. [1] attribute this dominance to the open-source ecosystem and broad device support.

The Android Open Source Project (AOSP) [60] provides a platform for developers to create custom distributions and compatible applications. AOSP publishes the source code under the Apache License 2.0. This license allows manufacturers to modify the operating system for specific hardware and branding requirements. These modifications generate a diverse device market [1]. We distinguish between "official" Android devices and "Android-based" systems. Official devices comply with the Android Compatibility Definition Document and pass the Compatibility Test Suite [61]. This certification grants access to Google Mobile Services and the Google Play Store [62]. Conversely, Android-based systems utilize AOSP but operate without these proprietary components [1]. For users, this means a high degree of customization and flexibility. They are able to personalize their devices with widgets, themes and third-party launchers, while also having access to an extensive app ecosystem offered via the central Google Play Store [62].

2.1.1 Architecture

The Android operating system has a multi-layered architecture [63] in form of a stack, as shown in Figure 2.1. The lower layers of the Android architecture are closer to the hardware and provide essential system-level services. In contrast, the higher layers are more abstract and offer a richer set of APIs and tools for developers to create and interact with Android applications.

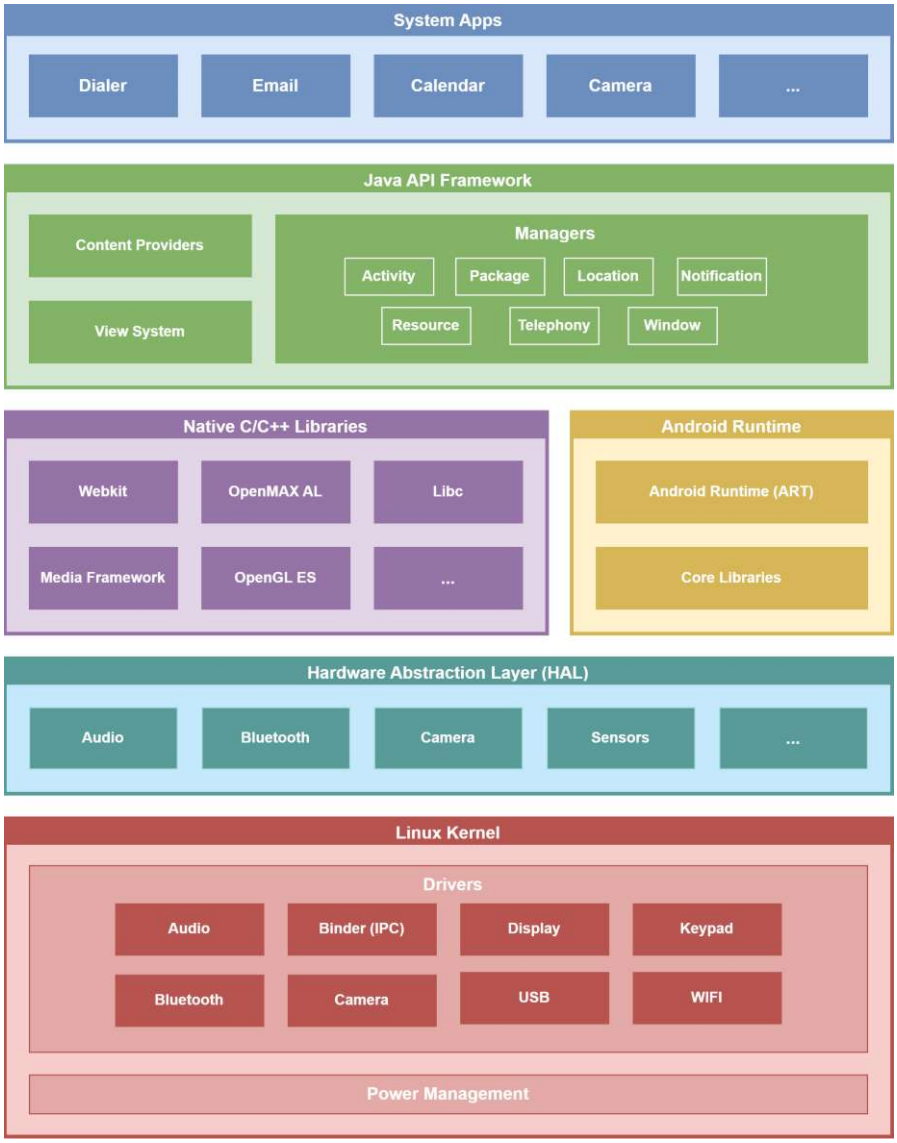

Figure 2.1: Android platform architecture [63]

Linux Kernel

The Linux kernel forms the base of the Android architecture and provides basic system services such as process management, memory management, and device drivers [64]. The kernel is open-source software with a large community of contributors who ensure constant updates and improvements [65]. Due to this fact, it is known for stability and support for a wide range of hardware architectures and devices [63]. In addition, the kernel provides a strong security model with functions such as file and user-based permissions, secure inter-process communication (IPC), and process isolation [66]. Android builds its security architecture on these mechanisms.

Hardware Abstraction Layer

The Hardware Abstraction Layer (HAL) [67] resides directly on top of the kernel. Meike and Schiefer [2] explain that the HAL abstracts hardware-specific details to provide a standardized interface between the Android framework and hardware components. Android applications use these APIs to interact with the device's hardware. The HAL has several modules, each of which implements an interface for a specific type of hardware component, such as the camera module [68]. This architecture allows device manufacturers to develop and customize hardware drivers and firmware without modifying the Android framework [2]. This approach enables faster development and testing of new hardware components and facilitates the integration of third-party hardware.

Native daemons and libraries

This layer consists of low-level system processes and shared libraries that are written in C or C++ and provide core system functionality [2]. Examples of native daemons include the zygote process, which is responsible for launching Android applications, and the surface flinger process, which manages the display and graphics rendering [69, 70]. Some examples of shared libraries are the Bionic libc library, which provides standard C programming functions optimized for Android, and the libmedia library, which supports multimedia operations such as audio and video playback [2]. These daemons and libraries interact directly with the kernel and do not depend on a userspace-based HAL implementation [71].

Android Runtime Environment

The Android Runtime Environment is responsible for executing Android applications. It performs the translation of the app's bytecode into processor-specific instructions that are executed by the device's runtime environment [72]. We discuss the historical Dalvik Virtual Machine in Section 2.1.2 and take a closer look at the current Android Runtime in Section 2.1.3.

Java Core Libraries

Android has its own set of Java core libraries that were originally derived from the Apache Harmony project [73] and includes classes and interfaces for tasks such as data manipulation, file handling, networking, and multithreading [2].

System Services

System services are modular components that implement fundamental Android features such as managing the device's connectivity, handling notifications, managing the device's power, and providing access to system resources [2].

Java API Framework

The Java API Framework is a collection of classes, interfaces and precompiled code that expose the feature set of the operating system [71]. Although Kotlin is the preferred language for Android development, the framework layer primarily consists of Java code. This layer provides the building blocks to create applications and includes components to interact with activities, system services, broadcast receivers, and content providers [74].

Applications

The applications form the top layer of the Android architecture. This layer consist of pre-installed system applications, such as the phone dialer and web browser. It also includes third-party applications that users install from the Google Play Store or alternative marketplaces [63]. Section 2.2 discusses Android applications in detail.

2.1.2 Dalvik Virtual Machine

The Dalvik Virtual Machine (DVM) is the original, but discontinued execution environment that was in use until Android version 4.4 [75]. Elenkov [3] explains that the introduction of the DVM represented a significant departure from the conventional Java Virtual Machine (JVM), as it focused on the specific needs of mobile computing such as efficient processing and power conservation. Chell et al. [4] describe the DVM as a register-based architecture, whereas the JVM operates on a stack-based model. The register-based architecture requires fewer instructions to perform equivalent operations, which minimizes the CPU overhead associated with instruction dispatch.

Build tools compile Android applications written in Java or Kotlin into Java bytecode, which they subsequently translate into Dalvik bytecode [3]. To further enhance execution speed, the DVM incorporated Just-In-Time (JIT) compilation starting with Android 2.2 [76]. This mechanism dynamically compiles frequently executed bytecode into native machine code at runtime. JIT optimizes these "hot" code paths to use system resources more efficiently [77]. However, this approach introduces runtime overhead, which increases battery consumption and application startup latency.

2.1.3 Android Runtime

The Android Runtime (ART) replaced the DVM as the default runtime starting with Android 5.0 [75]. ART retains backward compatibility with existing DEX bytecode and ensures that most applications developed for Dalvik work under ART. It builds upon DVM concepts but aims to enhance performance and efficiency [78]. To achieve this, it addresses JIT limitations by adopting Ahead-Of-Time (AOT) compilation [72].

During app installation, the `dex2oat` tool performs AOT compilation to transform DEX bytecode into native machine code [79]. The resulting output uses the Of-Ahead-Time (OAT) format. Some system libraries still use the `.oat` extension, while user applications standardize on the `.odex` extension. These artifacts function as ELF binaries and wrap the identical internal OAT structure regardless of the specific file suffix. Android 8 separates the compilation output into `.odex` files containing native code and `.vdex` files holding uncompressed DEX data and validation metadata. This separation allows the runtime to update optimized code without re-extracting the original bytecode.

Android 7 introduced a hybrid engine that combines profile-guided AOT, JIT, and interpretation to optimize resource utilization [79]. Furthermore, ART provides significant improvements in garbage collection and offers advanced tools for app development, testing, and debugging [80].

2.2 Android Applications

Android applications, commonly referred to as apps, are primarily developed using Java, Kotlin, C++ and increasingly, other languages like Dart (Flutter) that allow for cross-platform development [81]. We classify applications as ‘native applications’ when they are developed using the Android SDK and languages such as Java and Kotlin, since they are specific to the Android platform and offer the best performance and usability. Web applications are developed using web technologies such as HTML5, CSS and Javascript, but are not installed on the device. Another option is ‘hybrid applications’, i.e. a web application that is packaged in a native container and subsequently also has access to device functions. In this thesis, we focus primarily on native apps and differentiate between the following installation types [71]:

- *Privileged app:* System or privileged applications are pre-installed on devices and are integral for basic functions such as the dialer or camera app. An app created using a combination of the Android and system APIs. These apps must be preinstalled as privileged apps on a device.
- *Device manufacturer app:* An app developed by utilizing a combination of Android API, System API, and direct access to the Android framework implementation. A device manufacturer may directly access unstable APIs within the Android framework, so these applications must be pre-installed on the device and can only be updated when the device’s system software is updated.

- *User-installed Apps:* An app created solely using the Android API and was either installed from Android's application market *Google Play* [62] or 'sideloaded', i.e installed from external sources and third party vendors.

The first two app types can be simply summarized as system or preinstalled applications and are typically included in the OS image. They are normally installed in the `/system` partition, but from Android 9 and higher there are also `/product` and `/vendor` as additional partitions [82].

The `/system/app` directory holds standard system applications, while `/system/priv-app` contains privileged system apps with elevated permissions that require deeper access to system functionality [82]. These locations are part of the read-only section of Android's file system to protect system apps from modification or removal by standard users. The most common location for user-installed apps is the `/data/app` folder. Legacy Android versions also utilized the `/data/app-asec/` folder for apps on external storage and `/data/app-private` for protected internal applications [3]. However, modern Android versions largely deprecate these specific directories due to changes in Android's app storage management practices.

In the current and newer versions of Android ($\geq$ API level 29), the approach to app storage has been refined with concepts such as Scoped Storage [83], which aims to improve user privacy and security by granting apps limited access to external storage unless the user explicitly grants permission for wider access.

2.2.1 Android Package Kit

The Android Package Kit (APK), typically known by its file extension `.apk`, is the file format used for distribution and installation of mobile apps. APK files are essentially ZIP archives that bundle application code, resources, and assets [84].

It contains the following key components [85]:

- `AndroidManifest.xml`: The Android manifest contains meta information and configuration of application components. This includes the package name, version, permissions, services and the components of the application such as activities, services, broadcast receivers and content providers.
- `res`: The `res(sources)` directory holds all non-code resources such as images, strings, layouts, and user interface components. These resources are accessible from the code through automatically generated identifiers, allowing for flexible design and localization without altering the underlying codebase.
- `assets`: In comparision to the `res` directory, this location is best suited to store raw files such as text files, configuration files, or other data formats not natively supported by Android's resource system. Files can be accessed through an `AssetManager`, which provides a way to load them as stream objects.

- `lib`: This directory stores compiled native libraries, typically written in C or C++. Google Play policies mandate support for 64-bit architectures [86]. Developers must include `arm64-v8a` for physical devices and optionally `x86_64` for emulators. Legacy 32-bit architectures, such as `armeabi-v7a` and `x86`, are still permitted but require a corresponding 64-bit library to be compliant.

- `classes.dex`: This file contains the compiled Java or Kotlin code (in the form of Dalvik bytecode), which is executed on the Android Runtime. When an application grows in size, the single `classes.dex` file might exceed the limit of the DEX format, which can handle only a certain number of methods and fields. To manage this, the build tools split the bytecode across multiple DEX files (e.g. `classes.dex`, `classes2.dex`, etc.).

- `META-INF`: This directory stores metadata and verification data for the legacy v1 signing scheme [87]. The `MANIFEST.MF` file enumerates all archive files and their cryptographic hashes. This list includes `AndroidManifest.xml` and `classes.dex`, but excludes the signature files themselves to prevent circular dependencies. The `CERT.SF` file stores the digests of the `MANIFEST.MF` entries. The `CERT.RSA` file holds the signer's public key certificate and the digital signature of the `CERT.SF` file. While modern Android versions use the APK Signature Block, these legacy artifacts remain to ensure backward compatibility [88].

2.2.2 Dalvik Executable Format

The Dalvik Executable Format (DEX) format serves as the container for compiled application code and metadata on the Android platform [89]. The file structure consists of a header, identifier lists, and a data section.

The header stores essential metadata, such as the file version and checksums for integrity verification [89]. It also includes a map list that functions as a table of contents for the file. A key feature of the format is the shared constant pool, which eliminates redundancy by storing identifiers for strings, types, and methods in unique lists. For instance, if multiple classes reference the same string, the file stores the string only once and references it via an index.

Class definitions link these identifiers to the actual class body located in the data section [89]. This section defines the implementation details for every method, such as memory requirements and exception handling logic. It also contains the sequence of raw bytecode instructions. We specifically target this instruction stream to apply our opcode permutation algorithm. Appendix B.1 provides the complete DEX file layout.

2.2.3 Dalvik Bytecode Format

The Dalvik Bytecode Format defines the instruction set executed by the DVM and ART [90]. Build tools convert source code into this operational code and package it within DEX files. Dalvik bytecode includes a wide array of instructions optimized

for common operations on mobile devices. The instruction set includes operations for arithmetic calculations, type conversions, and control flow management. It also provides specific opcodes for object manipulation, method invocation, and array handling.

We exemplify the bytecode structure using the binary `add-int` instruction [90]. This operation performs a 32-bit integer addition of values from two source registers and stores the result in a destination register. The instruction utilizes the `23x` format layout and occupies two 16-bit code units. The first eight bits encode the opcode value, such as `0x90`. The next eight bits specify the destination register, while the remaining 16 bits define the source operand indices. ART validates the bytecode structure prior to execution to enforce type safety and prevent illegal operations [78].

2.2.4 Native Development Kit

The Android Native Development Kit (NDK) [91] is a set of tools that allows developers to implement parts of their Android applications using native-code languages such as `C` and `C++`. Native code runs directly on the processor and therefore bypasses the runtime overhead, which can lead to significant performance improvements for heavy computational tasks or real-time processing. Native code is also less susceptible to traditional Java decompilers, which adds an additional layer of complexity to reverse engineering efforts.

The NDK is fully supported by Android Studio and includes various build systems such as `CMake` or `ndk-build` [91]. These build systems enable the compilation of native code into libraries for Android applications. They support the primary CPU architecture families used in modern devices such as `ARM` and `x86`. This requires developers to manage builds and testing across these architectures to ensure broad compatibility.

The Java Native Interface (JNI) [92] is part of the Java platform and independent of the NDK, but plays a critical role in bridging the gap between JVM code and native code when using the NDK. JNI allows Java code to include native methods, where developers can call them as if they were regular Java/Kotlin methods. For example, native code can create new Java objects, call Java methods, and read or write Java fields. Native methods can also throw and check exceptions to the Java side, allowing for integrated error handling between the two environments.

2.3 Android Security

Android's security framework relies on several core components that safeguard user data and system integrity [66]. Many security components are provided by the Linux kernel, but some additional mechanisms are implemented by Android itself. Two examples for Linux kernel security mechanism are filesystem permissions [66] and Security-Enhanced Linux (SELinux) [93]. Filesystem permissions regulate access to files and directories. This allows applications to interact only with authorized resources. SELinux enforces mandatory access control at the kernel level and implements strict security policies across the system.

Android builds upon these kernel capabilities to implement its own security mechanisms. For example, the application sandbox [94] isolates each application so that processes run independently and cannot access each other's data without explicit permission. Application signing enables the Android system to verify the authenticity of apps and detect any tampering, ensuring that updates come from the same developer [87].

Android's permissions model governs app access to sensitive data and system features, while encryption mechanisms such as full-disk encryption (FDE) and file-based encryption (FBE) protects user data by encrypting storage either entirely or on a per-file basis [95]. To ensure the integrity of the device's software at each stage of the boot process, it is verified by the Verified Boot security feature [96].

Hardware-backed security features, such as the Trusted Execution Environment (TEE) [97] and the Android Keystore System [98], securely manage cryptographic keys and sensitive operations. Biometric authentication [99] offers secure and convenient user verification methods such as fingerprint and facial recognition.

We have given a brief overview of some major components of the Android security model and will now discuss a subset of these components in more detail.

2.3.1 Application Sandbox

Android's application sandbox ensures that applications cannot access the data or memory of other applications [94]. Figure 2.2 illustrates the sandbox mechanism, where each application runs within its own sandbox isolation boundary. This boundary encapsulates the application's private memory and storage.

Mayrhofer et al. [5] state that the fundamental mechanism behind Android's sandboxing is Discretionary Access Control (DAC), similar to Linux desktop systems. In Linux, each user is assigned a User Identifier (UID) and can be added to groups that are identified by Group Identifier (GID). Access to resources, e.g. a file, is regulated by these UIDs, and permissions can be altered by their owner. On Android, UIDs are not assigned to physical users because smartphones are usually single-user devices. Instead, the system automatically assigns a unique UID to each application at installation time. Each application runs in its own Linux process under its assigned UID and has access only to its designated data directory in the filesystem.

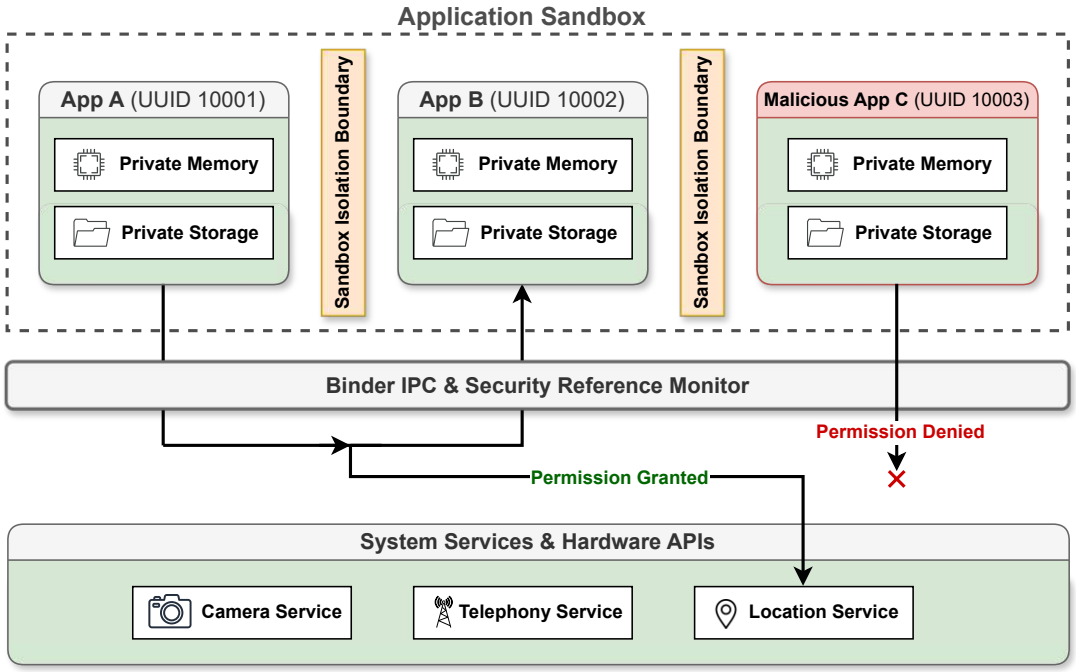

Figure 2.2: Application Sandbox and IPC Architecture

The filesystem permissions are enforced by the kernel and restricts the application from accessing other directories or modifying system files [94]. Attempts to access unauthorized files result in permission errors, as the kernel mediates all file operations based on UID and GID ownership and the associated permission bits (read, write, execute). This means no other application has permission to read or write its data. As a result, Android applications are sandboxed (isolated) at both the file and process levels, with process-level sandboxing achieved by running each application in a dedicated process and file-level sandboxing resulting from each application having its own isolated data directory. Consequently, all legitimate system service or inter-application interactions must be routed through the secure *Binder IPC* mechanism shown in the figure.

Android 4.2 introduced multi-user support, which allows multiple users to share a single device while maintaining data isolation [100]. Since the Linux kernel supports only a single numerical range for UIDs, Android cannot assign the same UID to an application installed by two different users, as they would share file permissions.

To solve this, Android assigns a new effective UID, the Android ID (AID) for every application instance [5]. The AID is calculated using a numerical composition: the system enforces a fixed offset per user and multiplies the AID by this offset and adding the application's base UID [3]. For example, if an app has a base UID of 10001, it runs as UID 10001 for the primary user (User 0) but as UID 1010001 for a secondary user

(User 10). This arithmetic separation ensures that even identical applications installed by different users possess unique UIDs and places them in distinct sandboxes.

2.3.2 Application Permissions

Android applications have access restrictions for device files and resources because of the platform's sandboxing system [101]. To gain more access rights to resources and functionality, so called *permissions* can be granted to apps. According to Six [6], the permission model is essential to protect users by controlling application access to sensitive data and device functions, such as the camera, location, contacts and network. Permissions are declared in the app's manifest file and depending on the type of permission may require explicit user approval either at install time or during runtime. This gives users the ability to understand and control what data or features an app can access.

Android has different types of permissions based on the sensitivity of the resource [6]:

- *Install-time permissions* give limited access to data or enable it to perform actions with minimal impact on the operating system or other applications. As the name suggests, permissions of this type are requested before the user installs the application. There are two sub-types of install-time permissions:
  - *Normal permissions* allow actions that extend beyond an app's sandbox, but have minimal risks to user privacy or device security. Since these permissions are low-risk, the Android system automatically grants them when the app is installed, without requiring explicit user approval. For example, the ability to access the internet, Bluetooth connection and device vibrations are such permissions.
  - *Signature permissions* are granted only if the app requesting the permission is signed with the same certificate as the app that declared the permission. This type of permission enables secure interaction between apps from the same developer and allows data or functionality to be shared without requiring sensitive user permissions.
- *Runtime permissions* (or *dangerous permissions*) allow an application access to private user data or let it perform restricted actions that substantially affect the system and other apps. Unlike install-time permissions, which are granted automatically when the app is installed, runtime permissions require explicit user approval during app usage. When an app needs to perform an action that requires a runtime permission, it must first check if the permission is already granted. If it is not granted, then the app prompts the user to approve the specific permission. The ability to access the camera, contacts or location are examples of runtime permissions.

- *Special permissions* allow the access to powerful system features and are typically used by system or privileged apps. They can only be defined by the Android platform and original equipment manufacturer (OEM). The modification of system settings and displaying content over other apps are examples of special permissions.

2.3.3 Application Signing

Android application must be digitally signed by their developers before it can be installed and run on a device or distributed through app stores [87]. This digital signature serves as a unique identifier for the developer and enables the Android operating system to verify that the application has not been tampered with since it was signed. In other words, it ensures the integrity and authenticity of applications.

Public-key cryptography (or asymmetric cryptography) is used for the application signing [6]. The developer holds the private key of the certificate to sign the application, while the public key is visible to everyone and is used to verify the signature. The public certificate (public key and metadata) is included in the `META-INF` directory of the APK. It is also possible for developers to sign two or more applications with the same key to assign them a shared UID, which allows the applications to share files and run in the same process.

When a user installs or updates an app, the *Android Package Manager* [102] validates the APK's signature using the embedded certificate and guarantees that updates are from the same, original developer. On an update, the verification compares the certificate of the currently installed application with that of the updated version and confirms that the APK has not been altered. A certificate authority is not required to sign the certificate, self-signed certificates are sufficient [6]. In general, therefore, there is no guarantee that the signed code originates from a trustworthy developer nor that it is safe to execute [3]. However, application signing allows the Android OS to distinguish system applications from regular applications by signing them with the same certificate as the Android firmware.

Although the official recommendation is to sign apps with all APK scheme versions (v1, v2, v3) for maximum compatibility of older Android versions, Wang et al. [7] found a number of issues for the v1 scheme. However, Google introduced the Android App Bundle (AAB) publishing format to optimize the distribution and delivery of applications [103]. The key difference between the APK and the AAB process is that with APKs, developers build and sign the final application package themselves before distributing it directly to users or app stores [84]. In the case of AABs, developers upload a signed bundle that contains all compiled code and resources to Google Play, which then generates and signs optimized APKs tailored to each user's device configuration.

For this purpose, developers first need to sign the AAB using a private *upload key* [103]. This key verifies their identity when uploading to Google Play and can be renewed if necessary. Google Play processes the AAB and generates the optimized APKs for different device configurations such as screen resolutions, processor architectures and language.

Those optimized APKs are then signed with the *app signing key*, which is securely managed by Google and remains constant to ensure that app updates are trustworthy. Afterwards, the app goes through Google's review process and is published on the app store. Since August 2021, Google Play requires the use of the AAB format for all new apps uploaded to the platform.

2.3.4 Security-Enhanced Linux

Security-Enhanced Linux (SELinux) [93] is a Linux kernel security module that supports the enforcement of mandatory access control (MAC) policies. Unlike discretionary access control (DAC), which grants access based on user ownership or permissions, SELinux introduces an additional layer of security by defining policies that strictly govern interactions between processes, users, and files. This approach restricts processes to only the resources and actions explicitly defined in their policies, which reduces the risk of unauthorized access. Furthermore, Negus [8] explains that SELinux is an extension for DAC and not a replacement, since DAC rules are checked first and if access is allowed, only then are the MAC policies checked.

Since Android 5.0, the system fully enforces SELinux policies, blocking any process action outside its allowed domain [75]. Smalley and Craig [9] highlight that this adaptation confines privileged daemons and strengthens application sandboxing at the kernel level to prevent privilege escalation.

2.4 Mobile Application Reverse Engineering

Reverse engineering is the process of analyzing a program or system to understand its components, functionality and operation without access to the original source code or documentation. In the context of Android applications, reverse engineering focuses on dissecting APK files to examine the application's code, resources and assets.

Benign actors, such as security analysts and researchers, perform reverse engineering to investigate the behavior of malware, the impact on privacy or the risks of certain software components. In contrast, malicious actors try to exploit vulnerabilities or bypass security measures. For example, attackers may circumvent license checks or extract sensitive information such as API keys, encryption keys or user credentials that could be used to gain unauthorized access or launch further attacks.

Code obfuscation is a countermeasure to reverse engineering that developers use to protect their mobile applications. To evaluate our virtual machine-based obfuscation, we need to know what reverse engineering techniques are typically employed under Android, how they work and if they work against our obfuscation. Therefore, we delve into the common methods of static and dynamic reverse engineering in the following sections.

2.4.1 Static Reverse Engineering

For static reverse engineering on Android, the APK file is unpacked and inspected without actually running the application [4]. Java and Dalvik bytecode contain more information than machine code because they preserve higher-level constructs such as classes, methods and interfaces from the original source code. This allows for an inspection of its application code and its transformation into a more human readable code.

Faruki et al. [10] show that in addition to the application code, embedded resources such as images, strings or layout files can also be retrieved and consider libraries or third-party dependencies to be a high-value target to identify known vulnerabilities or risks. For example, an app's `AndroidManifest.xml` can disclose sensitive components or improperly exposed entry points, while resource files may provide clues about the app's structure and dependencies.

Decompilation

Decompilation is the process of reversing the actions of a compiler. It takes an executable program, which has been compiled into machine-dependent code, and seeks to reproduce code in a high-level language that replicates the behavior of the original executable. More specifically, Android decompilers transform an APK's compiled DEX bytecode back into a easily readable type of Java code, regardless whether the application was written in Java or Kotlin [4]. This translation does not always result in an exact replica of the original source code. However, the reverse-engineered code often provides enough detail for an analyst to gain a thorough understanding of an application's logic and control flow. Mauthe et al. [11] found in their empirical study that the most common causes of failed decompilation are advanced obfuscation techniques and technical limitations of decompilers. For example, they claim that complex control flow or deep nesting levels (e.g. inheritance, inner classes, etc.) cause many decompilers to fail from internal resource exhaustion.

Android disassemblers perform a subset of the high-level language reverse operation. They translate the Dalvik bytecode into Smali [104], a low-level assembly-like language specific to the Android platform. Unlike decompilation, which aims to reconstruct high-level code, disassembly offers a step-by-step representation of the app's operations, which makes it useful for pinpointing exact instructions and identifying low-level vulnerabilities or examining heavily obfuscated code that might resist high-level decompilation [4].

Repackaging

Repackaging is the modification and redistribution of an existing application. This includes everything from legitimate customizations to unauthorized or malicious activity. Merlo et al. [12] explain that attackers often download popular apps, reverse engineer them (typically by decompiling) and make various modifications. These modifications may include rerouting app earnings, injecting malicious code or embedding unwanted

advertisements. Once modified, the app is repackaged and distributed as a cloned or plagiarized version.

This practice is particularly prevalent within the Android ecosystem, where apps can be self-signed and uploaded to the official store or shared on third-party stores through sideloading. Although repackaging attacks are a concern in unofficial app stores, they are rather difficult to carry out on Google Play today due to its robust security measures and policies [87]. Specifically, Google Play's combination of application signing, app review processes and continuous monitoring with Play Protect creates a strong defense against such attacks [105].

Repackaging is also beneficial for security researchers, as it allows them to modify unknown applications to better understand their functionality. For example, researchers could disable certain anti-reverse engineering mechanisms and add debugging or logging functionalities to monitor the app's behavior in a controlled environment.

According to Glanz et al. [13], it is not trivial to detect repackaged apps due to several factors. Obfuscation techniques make the repackaged app appear substantially different from the original, even when its functionality remains largely unchanged. Another problem is that a significant part of an app's codebase may consist of third-party libraries, which are commonly used across multiple apps. As a result, the presence of similar library code does not necessarily indicate repackaging, but leads to high rates of false positives.

Tools for Static Reverse Engineering

A variety of tools are available for performing decompilation and disassembly on the Android platform, with the most widely used ones introduced below:

apktool [106] is an open-source command-line tool designed for reverse engineering APKs. What sets *apktool* apart from other reverse engineering tools is its ability to decode and rebuild Android apps while preserving the original resources and structure.

JADX [114] functions as a Dex-to-Java decompiler that converts Dalvik bytecode into readable Java source code. This tool facilitates the inspection of application logic and manifest configurations through both command-line and graphical interfaces.

Ghidra [115] provides a comprehensive software reverse engineering framework developed by the National Security Agency (NSA). It performs disassembly and decompilation across multiple architectures. In the Android context, Ghidra analyzes both the bytecode and native libraries to reconstruct control flow graphs and high-level logic.

2.4.2 Dynamic Reverse Engineering

Static reverse engineering reaches its limitations when targeting heavily obfuscated or encrypted code. Protection techniques such as class encryption or advanced control flow obfuscation hinder static analysis or prevent it altogether [4]. In such cases, dynamic

reverse engineering allows analysts to observe the application behavior during execution and bypass static protection mechanisms. Sutter et al. [14] list behavior such as system call monitoring, memory inspection, and execution flow tracing. Dynamic analysis overcomes obfuscations by identifying decrypted strings or dynamically loaded code in memory during execution. This shift to runtime observation enables a deeper understanding of the app's logic, even in the presence of advanced obfuscation.

Debugging and Emulation

Debuggers allow analysts to suspend execution, inspect memory, and modify register values. On Android, the Java Debug Wire Protocol (JDWP) facilitates debugging at the Dalvik bytecode level [3]. However, for analyzing the ART itself or native libraries, native debuggers such as the Low Level Debugger (LLDB) [107] or GNU Debugger (GDB) [108] are necessary. Sharif et al. [15] show that native debugging is particularly relevant for virtualization obfuscation, as it allows attackers to step through the interpreter's execution loop and observe how the custom instruction set is processed.

Emulators, such as the Android Emulator (based on QEMU [109]), simulate the hardware environment. They allow the execution of Android operating systems without physical devices [110]. For reverse engineering, emulators offer full system visibility, including the ability to dump the entire system memory or trace kernel-level instructions.

Dynamic Code Instrumentation

Dynamic code instrumentation refers to the process of modifying the instructions or behavior of a program during runtime. In other words, a dynamic code instrumentation tool interacts with a running process without requiring source code modifications.

Dresel et al. [16] show that dynamic code instrumentation works by hooking or injecting instructions into a running program. Hooking is a technique used to redirect the normal flow of execution to custom code or other specific points in a program such as function calls, system calls or events. In comparison to debugging, the main advantage is that the program does not have to be stopped at breakpoints, which would often disrupt its normal behavior and timing. This makes it ideal for bypassing anti-debugging measures and analyzing complex, undocumented or obfuscated code.

Despite its powerful capabilities, dynamic code instrumentation leads to a performance overhead due to the introduction of additional runtime processing and may interfere with normal program functions, which could lead to unintended side effects or crashes [16].

Nevertheless, dynamic code instrumentation remains an important tool for reverse engineers because it provides real-time insights into program behavior and allows to bypass static obfuscation techniques.

Frida Toolkit

Frida [111] is a popular dynamic instrumentation toolkit, which supports multiple platforms including Android, iOS, Windows and Linux. In the context of Android, it allows the instrumentation of native code and Java bytecode through its powerful instrumentation core component that includes a JavaScript engine. Frida offers three primary modes of operation for dynamic instrumentation [112]:

- The *Injected* mode dynamically injects the Frida library into a running or spawning process. This mode is effective for quick, real-time analysis on rooted (or non-rooted) devices.
- The *Embedded* mode integrates the Frida gadget library into the application during development, which allows for persistent instrumentation and easy access to runtime behavior. This mode is ideal for controlled environments where embedding is feasible.
- The *Preloaded* mode automatically loads the Frida library into the target process at startup using techniques similar to `LD_PRELOAD`. This mode is suitable for automating instrumentation on platforms supporting library preloading.

Frida adopts a client-server architecture to communicate between the analyzing host (client) and the target device (server). In the case of Android, the Frida server runs on the Android device (or emulator) and injects and controls instrumentation scripts within the target application's process [113].

CHAPTER 3

Related Work

The security of Android applications is a constant race between offensive reverse engineering and defensive code protection. Steinböck et al. [17] explain that despite MASVS recommendations, standard hardening techniques are often inconsistently implemented and easy to bypass. In this chapter, we survey the current research landscape to pinpoint the specific defensive gaps that necessitate our virtualization-based approach. We begin in Section 3.1 by defining the threat model through an analysis of current static analysis and decompilation tools. To counter these threats, developers typically rely on traditional obfuscation techniques. However, Section 3.2 demonstrates that standard strategies, such as identifier renaming, often provide insufficient security against these threats. This inadequacy calls for stronger protection paradigms, which Section 3.3 addresses by analyzing state-of-the-art code virtualization frameworks. We identify architectural limitations in these approaches, particularly regarding performance overhead and native library dependencies, to define the research gap that our proposed opcode permutation mechanism addresses.

3.1 Static Analysis and Decompilation

Static reverse engineering represents the primary threat to Android application confidentiality. Researchers and attackers utilize these techniques to reconstruct high-level logic without executing the application.

Bytecode Disassembly. This fundamental static analysis technique maps binary opcodes to human-readable mnemonics. Smali [104] serves as the industry-standard assembler and disassembler for the Dex format. It strictly follows the Android Bytecode Specification to translate binary values into instructions (e.g., 0x90 to `ADD_INT`). *Apktool* [106] builds upon this foundation to reconstruct resource directories and the `AndroidManifest.xml`. These tools operate on the premise of a static instruction set.

As our implementation demonstrates, this dependency causes the disassembly process to fail immediately when it encounters our permuted opcode values.

Decompilation. Decompilers aim to lift bytecode back into Java or Kotlin source code. JADX [114] reconstructs the Control Flow Graph (CFG) to identify high-level constructs such as loops and conditionals. Mauthe et al. [11] analyze the robustness of decompilers against standard obfuscation techniques such as identifier renaming. They conclude that decompilers are resilient against simple renaming but fail when the underlying code structure deviates from standard compiler patterns. However, they do not account for virtualization techniques that completely hide the instruction stream. Our approach misleads those decompilers because the original instructions remain unknown to them.

Resilience against Static Detection. Sharma et al. [18] introduce MOSDroid, a framework designed to withstand obfuscation through semantic opcode analysis. They utilize stacked generalization ensembles to classify malware based on multisets of opcode sequences. This method achieves high accuracy against standard transformations. However, their approach fundamentally depends on the extraction of valid, recognizable instruction patterns. Aghakhani et al. [19] reinforce this limitation, demonstrating that machine learning classifiers based on static features often fail to generalize to unseen packing techniques because they overfit to specific packer artifacts rather than capturing the actual behavior.

Recent research shifts the focus of static analysis from Dalvik bytecode to compiled artifacts. A pipeline introduced by Bleier and Lindorfer [20] leverages the ART compiler to translate DEX bytecode into native ELF binaries (OAT files). They demonstrate that analyzing these AOT-compiled binaries using standard tools such as Ghidra [115] successfully recovers function boundaries that bytecode obfuscators attempt to hide. However, this method relies entirely on the standard ART compiler's ability to translate the input bytecode. Our virtualization approach neutralizes this threat because the standard ART compiler cannot parse our permuted instruction set. Consequently, it fails to generate valid OAT files and blocks this analysis vector.

3.2 Traditional Obfuscation Techniques

Guo et al. [21] classify Android protection strategies and find that renaming identifiers is the industry standard. They argue that renaming leaves program logic and API calls exposed, while control flow obfuscation frequently succumbs to compiler optimizations. Consequently, standard static transformations preserve the valid bytecode structure and provide insufficient protection against reverse engineering.

A large-scale comparative analysis by Kargén et al. [22] characterizes the usage of obfuscation in benign and malicious applications. The authors distinguish between trivial transformations, such as identifier renaming, and advanced techniques such as string encryption or control flow flattening. Their empirical data shows that simple renaming remains the dominant technique in the wild, whereas advanced obfuscation

appears in only a small fraction of apps. Furthermore, they challenge the assumption that obfuscation implies malicious intent, as legitimate developers frequently employ these methods to reduce package size and protect intellectual property. However, Bleier and Lindorfer [23] demonstrate that this widespread use of obfuscation severely hinders code similarity analysis. They show that aggressive transformations destroy the recognizable code structures required to identify benign code reuse. Consequently, obfuscated benign applications become indistinguishable from packed malware, which blinds similarity-based detection systems.

Niroshan et al. [24] track the evolution of obfuscation across 1.7 million applications over eight years. They attribute rising adoption rates to default ProGuard minification rather than intentional security hardening. The authors conclude that the ecosystem remains stagnant as developers ignore advanced virtualization in favor of standard renaming.

The resilience of static analysis features against standard obfuscation techniques is investigated by Molina-Coronado et al. [25] in the context of malware detection. The authors demonstrate that even trivial transformations, such as identifier renaming, significantly degrade the performance of learning-based classifiers. Their empirical results show that obfuscation reduces detection rates by approximately 25% because it distorts critical semantic features like API call chains and component names. Consequently, they argue that traditional static analysis features lack robustness against common obfuscation strategies and require complementary dynamic approaches.

Wong and Lie [26] categorize Android obfuscation strategies into language-based, full-native, and runtime-based techniques. They define traditional approaches as language-based obfuscation because these methods exploit standard facilities of the Java programming language, such as reflection, to hide method invocation targets. The authors distinguish this from runtime-based obfuscation, which subverts the integrity of the Android Runtime to alter standard code execution and method resolution rules. To defeat these techniques, they propose TIRO, a hybrid framework that employs an iterative "Target-Instrument-Run-Observe" approach. TIRO leverages static analysis to identify potential obfuscation sites and uses dynamic instrumentation within the runtime to capture execution data, which it feeds back into the static model to progressively deobfuscate the application.

3.3 Code Virtualization

Code virtualization transforms standard bytecode into a proprietary format interpreted by a custom engine. This technique isolates the program logic from the underlying platform. However, isolation can also be achieved through architectural features.

Software-based Virtualization. Shu et al. [27] propose SMOG, a virtualization system that randomly permutate standard Dalvik opcodes and executes them via a modified runtime environment. To support this, they replace the standard `libdvm.so` on the end-user device with a custom Dalvik Virtual Machine (DVM) containing a modified

C-based interpreter. While this method effectively defeats static analysis by hiding the instruction stream, it targets the deprecated Dalvik VM. We adapt this strategy for the modern Android Runtime (ART) (standard since Android 5.0). We address the challenges of integrating the obfuscation into ART's hybrid execution model [75]. Furthermore, we provide a working proof of concept, in contrast to their theoretical research.

A compile-time virtualization framework proposed by Zhao et al. [28] translates Dalvik bytecode into a custom instruction set. A native library, accessed via the Java Native Interface (JNI), interprets these proprietary instructions at runtime. While this strategy effectively hides the control flow from Java-based decompilers, it creates significant performance overhead due to the frequent context switching between the runtime and the native environment. Furthermore, the interpreter logic resides in a static native library. This makes it susceptible to binary analysis tools. Our approach differs fundamentally by modifying the Android Runtime itself. This eliminates the JNI bottleneck and integrates the obfuscation logic directly into the system's core execution engine.

Zhang et al. [29] investigate the internal diversification techniques of prominent virtualization obfuscators, including VMProtect [116] and Tigress [117]. The authors categorize these mechanisms into VM interpretation variants, bytecode organization strategies, and handler permutations. They demonstrate that high structural diversity in the virtual machine's design prevents deobfuscation tools from generalizing across different VM implementations. While their work focuses on inspecting native binary protection schemes, it validates the theoretical effectiveness of our diversification strategy. However, unlike the user-space virtualization layers they analyze, our approach integrates the opcode permutation logic directly into the Android OS. This enforces diversification at the system level rather than the application level.

CodeCloak, introduced by He et al. [30], is a virtualization framework designed to protect native Android libraries against repackaging. The authors implement a binary-level virtualization scheme that translates native ARM instructions into a custom stack-based instruction set executed by an embedded interpreter. To mitigate dynamic analysis, CodeCloak employs "time diversity". This mechanism generates multiple distinct mapping tables between virtual opcodes and their handlers. It dynamically selects a specific configuration during each execution cycle. While this approach effectively secures native code by shielding the original ARM instructions, it operates strictly within the application's native binary scope. Our obfuscation technique permutes the platform-agnostic bytecode rather than native machine instructions.

Hardware-based Isolation. Beyond software virtualization, execution environments can be shielded using hardware extensions. Quarta et al. [31] propose *Tarnhelm*, which leverages the ARM TrustZone to provide a transparent and confidential execution environment (TEE). It partitions the application to execute sensitive logic in the "secure world", effectively hiding it from the standard operating system. While hardware-backed isolation offers strong security guarantees, it relies on specific processor features.

In contrast, our approach operates entirely within the Android Runtime (software-level). This allows for greater portability across devices without specialized hardware dependencies.

Countermeasures. As a countermeasure to virtualization-based obfuscation, Xue et al. [32] introduce Parema, a framework designed to analyze and deobfuscate virtual-machine-based Android packers. They utilize dynamic binary instrumentation for runtime analysis and symbolic analysis to interpret and deobfuscate the bytecode. A similar hybrid method is employed by Graux et al. [33], combining dynamic monitoring and trace-based symbolic execution to generate CFGs of the application's behavior. You et al. [34] take advantage of the ART itself and use its built-in functionality for deobfuscation purposes of complex control flow obfuscations. To evaluate the efficacy of such dynamic testing, Bleier et al. [35] introduce *profile coverage*, a metric utilizing Android's compilation profiles to determine if analysis tools exercise the code paths actually used by real users. They show that traditional code coverage metrics often misrepresent the analysis quality compared to real-world usage, highlighting the critical reliance on accurate runtime execution data that our proposed opcode permutation aims to disrupt.

CHAPTER 4

Traditional Approaches to Android Code Obfuscation

Code obfuscation serves as a defense against reverse engineering, tampering, and other forms of attacks by obscuring the application's source code structure, logic or functionality.

Mobile apps are often exposed to on-device threats, including malware and unauthorized root access [118]. Code obfuscation must take these risks into account and implement techniques that protect sensitive code even if a device is compromised. However, we know that mobile devices often operate under more strict resource constraints than desktop environments such as power consumption. Therefore, the obfuscation has to consider the balance between security (e.g. root detection) and usability (installation size and execution efficiency).

Each mobile platform has their own set of application components and life cycles, which means that obfuscation techniques need to consider platform-specific elements such as Android manifest files, intents, and services without disrupting the app's functionality [81]. In addition, obfuscation strategies must take into account the nuances of different runtime environments, such as JIT or AOT compilation.

The last hurdle occurs with the distribution of obfuscated apps. App stores have specific guidelines and automated systems to detect potentially harmful behaviors. Obfuscation techniques should not trigger these safeguards erroneously, ensuring that apps remain compliant and available for distribution. There is a conflict of interest for the distribution platform here, as the distributors generally do not want to prevent legitimate developers from protecting their applications through obfuscation, but still has to enforce measures against obfuscated malware. Due to the intrinsic nature of obfuscation, it is often impossible to tell in advance whether an app contains malicious code.

In this chapter, we explore known obfuscation techniques and analyze their key characteristics to understand their strengths and limitations in protecting Android applications.

4.1 Identifier Renaming

Identifier renaming is a basic code obfuscation technique that aims to make source code harder to understand by changing the names of identifiers such as classes, methods, and variables to meaningless strings or characters. Zhang et al. [36] clarifies that this technique disrupts the readability of the code for humans but does not affect the execution or functionality from the perspective of the machine.

There are three common strategies for this obfuscation technique [36]:

- *Random Renaming:* This involves replacing original names with random characters or numbers.
- *Pattern-Based Renaming:* Uses a systematic approach to renaming identifiers following a specific pattern, which can help maintain some level of readability for authorized developers aware of the pattern. For example, DexGuard uses english alphabet characters [a-zA-Z].
- *Semantic Renaming:* A more sophisticated form that replaces names with others that are misleading or irrelevant to their function, creating confusion for anyone trying to reverse-engineer the code.

As an example, listing 1 contains the unmodified source code of a Java class with a simple method including two parameters.

```
1 public class Calculator {
2     public int add(int numberOne, int numberTwo) {
3         return numberOne + numberTwo;
4     }
5 }
```

Listing 1: Source code before identifier renaming

This source code gets transformed by Pattern-based Renaming, whereby identifiers are replaced by letters of the ascending alphabet. Listing 2 shows that the Java class originally named `Calculator` with a method called `add` is renamed to `A` with a method `a`, and parameters `numberOne` and `numberTwo` are shortened to `a` and `b`.

Listing 3 shows an example of Semantic Renaming, where the class name changes from `Calculator` to `StringUtil`, the method name from `add` to `concat`, and the parameter names from `numberOne` and `numberTwo` to `stringA` and `stringB`. These changes are made to mislead anyone trying to understand the code, suggesting that the class and method deal with string operations rather than mathematical calculations.

```java
public class A {
    public int a(int a, int b) {
        return a + b;
    }
}
```

Listing 2: Source code after pattern-based identifier renaming

The development of a semantic renaming strategy is complex and includes problems such as accurately understanding the context and functionality of each identifier to rename them semantically while maintaining the code's logical correctness. Another challenge is to ensure that renamed identifiers do not break the code, especially when reflection, serialization or external APIs are involved. Large Language Models (LLMs) could be a viable solution, because they possess advanced contextual understanding and natural language generation capabilities, which enables them to generate plausible, misleading identifier names while maintaining code functionality.

```java
public class StringUtil {
    public int concat(int stringA, int stringB) {
        return stringA + stringB;
    }
}
```

Listing 3: Source code after semantic identifier renaming

Identifier renaming can have a positive effect on reducing file size primarily due to the simplification of identifier names to shorter forms, which decreases the overall amount of data stored in the file [36]. For languages or environments that support reflection (e.g. Java and Android), renaming identifiers can break reflective calls, which rely on string names of classes or methods.

Android Studio supports code minification and shrinking primarily through the use of *ProGuard* and its successor *R8* [119]. These tools are integrated into the Android build system and are designed to optimize the application by reducing the size of the compiled APK and enhancing security through obfuscation. As an extra step, some interpreted languages and mobile web apps utilize minification to strip out all unnecessary whitespaces, new lines and comments. Listing 4 shows the single line of source code after minification.

In addition to identifier renaming, there exists Package Renaming as an complementary obfuscation [36]. Java and Kotlin programs are structured into packages, which are used to group related classes together and build a common namespace. Developers are

```
1 public class A{public int a(int a,int b){return a+b;}}
```

Listing 4: Source code after minification

free to choose the name of the package themselves, which can be leveraged to obscure the structure of the application. Identifier renaming targets the micro-level elements (variables, methods, classes), while package renaming targets the macro-level organization (packages and modules).

Listing 5 shows the package name before and after a random renaming strategy, where the sub modules identifiers were replaced by random letters and numbers.

```
1 package com.example.calculator;
```

```
1 package com.x3y9.z8k1;
```

Listing 5: Source code before and after package renaming

4.2 Control Flow Obfuscation

Control flow obfuscation modifies the logical execution paths of an application to obscure its true functionality from static and dynamic analysis [22]. This technique includes strategies such as junk code insertion, code flattening, and the incorporation of opaque predicates to disrupt the predictability of the control flow without altering the program's intended outputs.

These techniques are particularly effective against static analysis tools that rely on predictable control flow to generate accurate decompilations. By altering the execution paths and introducing non-trivial computational paths, control flow obfuscation can render automated decompilation tools less effective, as these tools may fail to correctly interpret the flow or may produce misleading results [37].

Balachandran et al. [38] explain that some control flow obfuscation techniques can be adjusted in complexity according to the security needs of the application. This allows developers to balance performance impacts against security gains. These obfuscations generally also do not require changes to the application's runtime environment or external dependencies, which makes them applicable across various software systems and platforms.

The disadvantage of control flow obfuscation is the introduction of additional computational steps into the execution path, which may lead to slower application performance and a higher risk of introducing functional errors [38]. This makes control flow obfuscation more difficult to implement than less invasive obfuscation such as renaming identifiers.

4.2.1 Junk Code Insertion

Junk code insertion involves adding non-functional, irrelevant code segments into the application's actual functional code [38]. This "junk" does not alter the intended outcomes or the external behavior of the program but serves to confuse and mislead anyone (or any tool) analyzing the code. For human reverse engineers, the added junk code increases the time and effort required to discern useful code from decoys.

The inserted code can range from no-operation commands (`NOP`) to more complex sequences that mimic functional operations but ultimately have no impact on the actual execution flow [38]. Other examples include redundant calculations, loops that do not affect outputs and function calls that perform unnecessary tasks.

Listing 6 shows an example of obfuscated source code derived from listing 1, where smaller snippets of junk code are integrated within functional logic rather than inserting large, obvious blocks. In the given `Calculator` class's `addNumbers` method, the snippet of code involving the `noise` variable serves as junk code. In line 4, we introduce a modulo scenario that might occasionally be true, but the addition and subtraction of the noise to variable `a` is irrelevant. The actual sum calculation happens in line 8. After that, we have a opaque predicate that is always true by the nature of how `sum` is calculated in line 9 and the encapsulated operation `sum = sum;` from line 10 contributes nothing.

```
1  public class Calculator {
2      public int addNumbers(int a, int b) {
3          int noise = a % 10;  // Plausible operation
4          if (noise > 5) {
5              a += noise;      // Appears functional but is irrelevant
6              a -= noise;
7          }
8          int sum = a + b;     // Actual sum
9          if (sum - b == a) {  // opaque predicate (is always true)
10             sum = sum;       // No operation
11         }
12         return sum;
13     }
14 }
```

Listing 6: Source code after junk code insertion

Opaque predicates are logical expressions or conditions in the code that always evaluate to a constant (true or false) but appear complex and non-trivial to an analyzer. For example, the expression `((x * x) % 2 == 0)` always returns `false`, but at the same time artificially branches the program and obfuscates the program's true logic from automated tools and manual inspectors.

Zobernig et al. [39] conclude that opaque predicates are useful under two conditions. First, the target program contains many conditional branches, such as constant comparisons ($x == c$) with "random" constants c, as well as variable comparisons ($ax + b == y$) with "random" constants a, b. This complexity makes it harder for automated deobfuscation techniques to distinguish between real and opaque predicates. The second condition states that generated dummy code blocks during obfuscation are indistinguishable from real code blocks in the program. They claim that without these conditions, opaque predicates become ineffective and easily removable.

Excessive or poorly implemented junk code renders obfuscation vulnerable to automated removal. For instance, You et al. [34] introduce Deoptfuscator, which analyzes the ART's internal intermediate representation to identify and strip obfuscation patterns that rely on global opaque variables. To withstand such advanced analysis, obfuscation must seamlessly blend non-functional code with the legitimate codebase. Effective strategies include employing global opaque variables to mimic state dependencies, ensuring style consistency, and using mathematical opaque predicates that appear non-trivial to static analyzers [38]. In other cases, developers may employ invalid bytecode sequences or 'traps' designed to cause decompilers to crash or generate incorrect output [34].

The potential impact on application performance comes from the additional junk code that is introduced by the obfuscation. More code means more processing, which can slow down the application, increase its memory usage, binary size and extend compilation times. Too little junk code might not provide enough obfuscation, while too much can make the application bloated and inefficient. Finding the right balance is essential to achieving effective obfuscation that protects the software while maintaining its operational efficiency and integrity.

4.2.2 Code Flattening

Code flattening or control flow flattening transforms the hierarchical structure of a program into a single, linear sequence. This is typically achieved by reorganizing the program's control flow into a series of blocks managed by a state machine or central dispatcher.

László and Kiss [40] decompose the original control flow into basic blocks containing elements such as nested loops, conditional branches, and function calls. A basic block consists of an instruction sequence where control enters at a single entry point and leaves from a single exit point. These basic blocks are then reorganized into the cases of a large switch statement and enclosed within a loop. A state variable controls the execution flow and determines the active case at any given time. The end of each switch case updates this variable to replicate the original program logic. These state updates replace direct control flow transitions, such as jumps between different parts of an if-else chain.

Listing 7 shows a simple method called `makeDecision` with three logical branches that prints out some text based on the numerical input.

```
1  public class Decision {
2      public void makeDecision(int input) {
3          if (input < 10) {
4              System.out.println("Input is less than 10");
5          } else if (input < 20) {
6              System.out.println("Input is less than 20 but not less than 10");
7          } else {
8              System.out.println("Input is 20 or more");
9          }
10     }
11 }
```

Listing 7: Source code before code flattening

Figure 4.1 shows the corresponding control flow graph of the `makeDecision` function and displays its simple hierarchical structure. Control flow representation is essential for reverse engineers because it shows how basic computational blocks are related to each other [38]. It simplifies analysis because it ensures that the execution path through the block is linear and predictable once entered. When reverse engineering a function, the control flow graph is particularly important for as it provides information about the different conditions to reach the basic blocks of the function and thus influences the decompilation process.

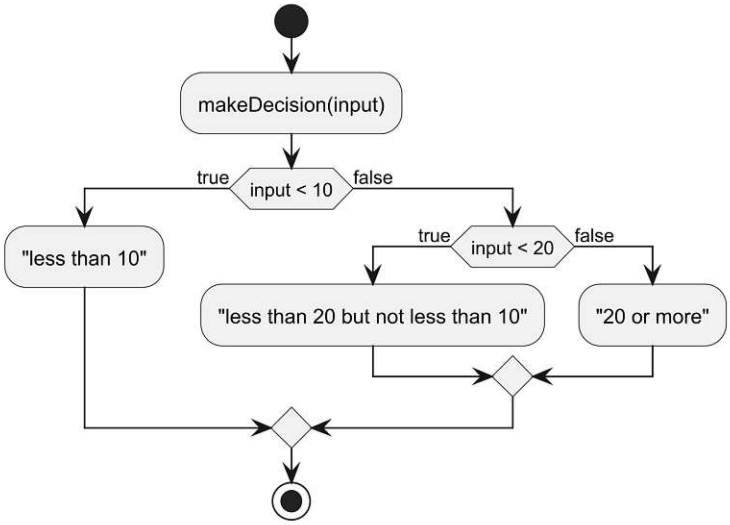

Figure 4.1: Control flow graph before obfuscation

Listing 8 shows the `makeDecision` code after an exemplary obfuscation with code flattening. A switch statement with a `int state` variable controls the logical flow, where `case 0` is used to set the state by using the conditions from the `if` statements. The switch is wrapped by a `while(true)` loop and the program is only able to escape by explicit `return` instructions. For example, `case 1` represents the first logical branch, prints out a message and uses the `return;` instruction to terminate the program.

```
1  public class Decision {
2      public void makeDecision(int input) {
3          int state = 0;
4          while (true) {
5              switch (state) {
6                  case 0:
7                      if (input < 10) {
8                          state = 1;
9                          break;
10                     }
11                     if (input < 20) {
12                         state = 2;
13                         break;
14                     }
15                     state = 3;
16                     break;
17                 case 1:
18                     System.out.println("Input is less than 10");
19                     return;
20                 // ... more cases
21             }
22         }
23     }
24 }
```

Listing 8: Source code after code flattening

Figure 4.2 shows the updated control flow graph after obfuscation and enables us to recognize the introduced complexity more easily at a glance. Code flattening removes the visual and logical patterns that are typically used by tools and analysts to understand program behavior. This linearization obfuscates the natural hierarchy of decision-making processes in the code. The use of a state variable to control flow introduces a level of indirection that makes it harder to predict and follow the program's execution path. Tracing through the code requires tracking the changes to the state variable, which can be non-trivial if the logic for changing the state is complex.

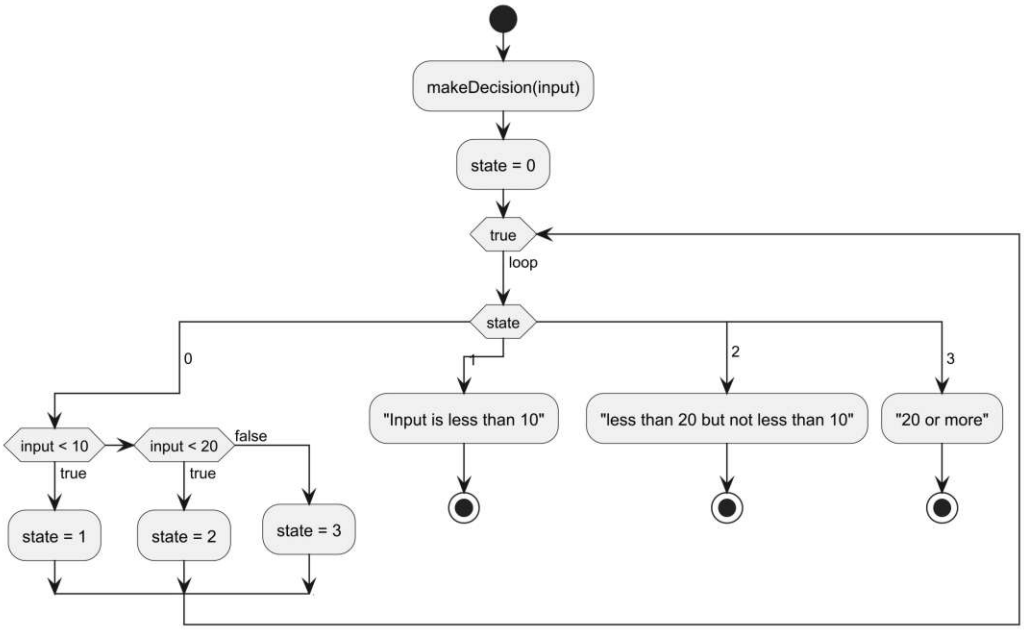

Figure 4.2: Control flow graph after code flattening

Another improvement to this obfuscation would be to fill the basic block of the default switch case with corrupted instructions to potentially stop tools from constructing the control flow graph [38].

4.3 Encryption

Encryption strengthens Android application security by transforming sensitive data and code into unreadable formats. This technique ensures that the information remains unreadable during static analysis and restricts interpretation to the runtime execution. Key management plays a critical role in this approach, as the security of the encrypted code depends entirely on the confidentiality of the keys. If an attacker extracts the decryption key, they are able to reverse-engineer the encrypted content.

The Android Keystore system and Trusted Execution Environment (TEE) offer robust hardware-backed protection for user data, but prove impractical for protecting the static decryption keys required for code obfuscation [97, 98]. Mass distribution of the application as a static asset requires decryption on the device immediately after installation. Since device-specific hardware keys cannot decrypt pre-encrypted bytecode without mechanisms such as remote key provisioning, obfuscation tools embed keys or decryption logic directly into the application to ensure autonomous execution.

This section addresses three encryption types common in the Android ecosystem: String Encryption, Class Encryption, and Resource Encryption.

4.3.1 String Encryption

String Encryption protects sensitive strings such as API keys and database credentials by converting them into encrypted formats [21]. This obfuscation technique typically employs symmetric encryption algorithms such as Advanced Encryption Standard (AES) to embed cipher text directly into the application source code. This prevents attackers from locating string constants via simple search operations. The application decrypts these strings at runtime using a managed key.

Listing 9 shows an example class with two string variables that store sensitive information in plaintext.

```
1 public class ExampleClass {
2     String url = "https://example.com";
3     String apiKey = "clearApiKey123"
4 }
```

Listing 9: Source code before string encryption

Listing 10 shows the obfuscated version, where encrypted values replace the literals and the `StringEncryption.decrypt()` method handles retrieval. This transformation secures the static APK file, but shifts the attack surface to the decryption method and the key management strategy.

```
1 public class ExampleClass {
2     String url = StringEncryption.decrypt("orXB+GtuIMKRKh7oMv0=");
3     String apiKey = StringEncryption.decrypt("5KpTgJ0LpUcfZyFgz1i=");
4 }
```

Listing 10: Source code after string encryption

The Android keystore is an effective approach to securely store keys in an encrypted loader architecture. However, developers also employ alternative key management strategies. Glanz et al. [40] identify 21 unique string obfuscation techniques used in real-world Android applications. These techniques involve various encryption, encoding, and transformation methods. Some implementations generate keys at runtime using device-specific identifiers to bind the decryption logic to the hardware. This hinders attackers from reusing retrieved keys across different devices. Other methods split the

key across multiple classes or resources and recombine them during execution to prevent discovery in a single location. Moving the key handling and decryption logic to native code using the Android NDK offers a extra layer of resistance compared to JVM bytecode. Fetching the key from a remote server at runtime is also a popular technique and keeps the key out of the application package entirely, but requires a secure and stable network connection.

4.3.2 Class Encryption

Class encryption secures application code by encrypting the bytecode of entire classes or methods [22]. This technique leaves the code unreadable during static analysis and provides an extra layer of security beyond standard obfuscation methods such as identifier renaming or control flow obfuscation.

Ilić and Dukić [41] propose a build-time encryption process where the application contains integrated decryption logic. The application dynamically loads (see Subsection 4.4.7) and decrypts classes in memory at runtime. This guarantees that only encrypted data resides within the application package. While this protects against decompilers, the decrypted code eventually exists in system RAM and remains vulnerable to memory dumping. Nevertheless, class encryption prevents static tampering and malicious code insertion. This security gain introduces performance overhead during class loading and increases build configuration complexity. Furthermore, the level of security depends on the encryption algorithm, its decryption keys and how the obfuscated class is loaded into memory.

4.3.3 Resource Encryption

Android applications include non-code assets such as images, audio, XML layouts, and configuration files [85]. Resource encryption prevents the exposure of these artifacts to protect critical implementation details. For instance, XML layouts expose application structure and component identifiers. Additionally, configuration files may leak API endpoints or internal flags. The motivation for obfuscation goes beyond software security, as it also protects intellectual property such as custom media resources, whose design requires significant investment. Attackers can easily extract and steal assets from unprotected APKs because they are simply packaged in a ZIP-based format.

Maiorca et al. [42] observe that most resource obfuscation targets elements stored in the `res/` and `assets/` directories. These tools compress assets in non-standard formats or encrypt them with known algorithms. The runtime environment decrypts or decompresses these assets on demand. Certain techniques also rename files and directories to meaningless strings to obscure the internal file structure. The `AndroidManifest.xml` is a high-value target as it holds critical metadata such as declared permissions, component definitions, and intent filters [120]. The obfuscation of the manifest is challenging due to its essential role in the application lifecycle and strict system validation during installation [42]. Despite the binary XML storage format, developers can encrypt

specific attribute values for runtime decryption. Another option is to restructure the `AndroidManifest.xml` in a way that breaks common decompilation tools.

4.4 Runtime Obfuscation

We define runtime obfuscation as a set of techniques that change the behavior of an application during execution to prevent static and dynamic analysis. Instead of just transforming static code, these techniques modify the runtime environment by dynamically decrypting, loading or interpreting code segments. Runtime obfuscations are therefore particularly effective against static decompilers and signature-based scanners, as the actual code is not present in a recognizable form within the APK. Furthermore, these techniques enable the application to dynamically adapt its behavior upon detecting debuggers, emulators, or tampered environments.

Wong and Lie [26] explain that runtime obfuscations are difficult to deobfuscate because the techniques may occur in one place and alter code execution in a seemingly unrelated part of the application. This means that the analysis can no longer be performed in isolated parts but must consider the application as a whole. To make it even harder, it is possible to protect applications against both static and dynamic reverse engineering attacks, when combining runtime obfuscations with static obfuscation, encryption and anti-debugging protections.

Nonetheless, runtime obfuscation has several disadvantages that impact performance, maintainability, and security. Debugging and maintaining an application with runtime obfuscation becomes significantly more difficult, since the transformed execution logic is not visible in the original source code. Runtime errors must therefore be avoided at all costs, as errors in reflective calls or dynamic loading mechanisms lead to ambiguous application crashes, unexpected behavior or security vulnerabilities. Another disadvantage are security systems such as Google Play Protect and antivirus scanners, which detect runtime modifications as suspicious behavior and trigger a security alert. For example, the modification of ART memory regions violates the Android security policy and unauthorized modifications are prevented since Android 8.0 (API 26+) in combination with SELinux [121].

Runtime obfuscation techniques are still susceptible to dynamic analysis, where attackers use debuggers, dynamic code instrumentation and memory dumping tools to inspect and modify sensitive information during execution. Haupert et al. [43] describe some protection mechanisms that can be implemented to support obfuscation techniques in their mission to hide sensitive information and prevent unauthorized access of the application. These protection mechanisms include anti-tampering, anti-hooking, anti-debugging, anti-emulator and anti-rooting. In literature, some sources classify these as part of runtime obfuscation techniques, since they employ “active“ protection mechanisms that obscures the program’s behavior when reverse engineering methods are applied. Other sources view them as distinct but complementary security layers that are reinforced by obfuscation.

In the following sections, we describe those protection methods as part of runtime obfuscation techniques and then discuss clearly classified runtime obfuscation techniques such as reflection, dynamic code loading and virtualization-based obfuscation.

4.4.1 Anti-Tampering

Tampering refers to the unauthorized modification of application code to bypass security features, alter functionality or inject malicious content [43]. Attackers who alter code during runtime obtain the same privileges as the application itself and once the code is compromised, other dynamic defenses can also be neutralized. To counter such threats, anti-tampering techniques verify code integrity and detect unauthorized runtime modifications. The application is then able to respond to the tampering threat and take action, e.g. terminate the process or execute code with misleading behavior.

A straightforward anti-tampering method is digital signature verification, where the app verifies its APK signature at runtime [43]. We know that Android performs signature verification during app installation (see subsection 2.3.3), but it does not recheck the signature during execution. Runtime digital signature verification solves this problem by retrieving the current app signature using the `PackageManager` API and comparing it to the original, trusted signature [102]. A mismatch between the retrieved signature and the expected signature strongly indicates unauthorized modification by a third party.

Another effective approach involves code integrity checks of application code and resources. These checks usually calculate cryptographic hashes of DEX files or native libraries and compare them with a precomputed, trusted hash [43]. Any mismatch between the expected and computed hashes indicates potential tampering.

The combination of these anti-tampering methods provides complementary protection, since code integrity checks detect internal changes to application components, while digital signature verification protects against external tampering such as repackaging.

4.4.2 Anti-Debugging

Anti-debugging techniques are designed to detect and disrupt software analysis or modification through debugging tools [4]. Common anti-debugging strategies include the utilization of system APIs to detect the presence of debuggers, the monitoring of certain exceptions and prevention through self-debugging.

Android has two possible debugging systems that must be taken into account for anti-debugging methods [43]. On one side, there exists Java-level debugger using the Java Debug Wire Protocol (JDWP) to bridge the gap to the Java VM. For example, listing 11 shows the usage of the `isDebuggerConnected()` function to check if the application is currently being debugged via the JDWP. If that is the case, then the application proceeds to terminate itself [122].

```
if (android.os.Debug.isDebuggerConnected()) {
    System.exit(1);  // terminate application
}
```

Listing 11: Anti-Debugging for JDWP

On the other side, we have debugger that utilize the `ptrace` [123] system call from the native Android Linux kernel to trace a process [43]. Anti-debugging methods for `ptrace`-based debuggers are typically done in native code (C/C++) and integrated into the application using the JNI, since direct access to `ptrace` is usually not available in JVM-based languages. For example, the Android application could invoke native code where `ptrace` is called on itself with the `PTRACE_TRACEME` request, as shown in listing 12. The entire technique relies on the fundamental rule of `ptrace` that a process can only be traced by one other process at a time. If the `ptrace` system call fails to trace its own application process, then we assume that a debugger is present and the application should immediately terminate. However, if this call is successful, then it serves as an active protection mechanism and any attempt by another process to attach itself as a debugger will fail.

```
#include <jni.h>
#include <sys/ptrace.h>
#include <stdlib.h>

JNIEXPORT void JNICALL
Java_com_example_AntiDebuggingUtil_preventDebugging(JNIEnv *env,
    jobject instance) {
    // attempt to make the current process traceable
    if (ptrace(PTRACE_TRACEME, 0, NULL, NULL) == -1) {
        // if ptrace fails, it indicates the presence of a debugger
        exit(1);
    }
}
```

Listing 12: Anti-Debugging for ptrace

4.4.3 Anti-Emulator

An emulator is a virtualized environment that mimics the hardware and software of a physical Android device [4]. Emulators are commonly used for reverse engineering Android applications because they provide a controlled environment to easily monitor and modify application behavior. Some techniques for detecting emulated environments include the analysis of file structures, system properties and hardware characteristics to identify differences from real devices.

Jing et al. [44] describe common detection methods used to identify virtual devices. One of these methods checks system properties such as `Build.FINGERPRINT`, `Build.MODEL`, and `Build.MANUFACTURER` for values like "generic" or "Emulator". Another approach is to scan for files such as `/dev/qemu_pipe` and `/system/bin/qemu-props`, which indicate the presence of an emulated environment. Hardware inconsistencies also provide reliable indicators of emulation. For example, physical devices typically have sensors such as accelerometers and gyroscopes, but are often absent in emulated environments. In most cases, several of these techniques are used in combination to achieve reliable detection.

4.4.4 Anti-Hooking

Li et al. [45] describe hooking as a runtime technique that intercepts function calls, method invocations, or system APIs by modifying pointers in the app's memory. This allows attackers to redirect the execution flow, alter method outputs or inject custom code, and is typically achieved through instrumentation frameworks. Since hooking operates at the memory level, static anti-tampering methods are ineffective against such attacks.

Anti-hooking techniques focus on detecting abnormal modifications to the app's execution flow and try to prevent unauthorized code injection [43]. One common method is runtime integrity checks, which periodically check the integrity of function pointers and memory regions to detect tampered code paths. Another common approach is to search the file system or memory for traces of known hooking frameworks such as Frida [111] or Cydia Substrate [124]. Code obfuscation also plays a key part in anti-hooking strategies, since it hinders attackers from identifying the target methods for hooking.

4.4.5 Anti-Rooting

Rooting an Android device is the process of obtaining the highest user privileges ("root") in the operating system [4]. It circumvents the application sandbox by allowing users to modify arbitrary (system) application files, remove pre-installed applications and perform operations typically restricted by the device manufacturer. Root access introduces significant security risks, but is often an requirement for reverse engineering tools. Root detection mechanisms are therefore implemented to protect against potential threats from rooted devices.

There are several common detection methods, one of which is to check for the presence of root granting applications such as `SuperSU` [125] or `Magisk` [126]. Another approach is to search for the binary file `su` in the system directories, as its presence indicates root privileges [4]. Additionally, verifying unauthorized read and write access to system partitions or properties can reveal potential rooting attempts. Finally, the Play Integrity API, which replaces the deprecated SafetyNet API, provides a standardized method for checking the security state of a device according to Google's integrity standards [127].

4.4.6 Reflection

Reflection is a mechanism that allows a program to inspect and manipulate its own internal structure at runtime [128]. It can be used to instantiate classes, access fields, and invoke methods dynamically without hardcoded references. Some obfuscations leverage reflection to hide explicit API calls. They replace direct invocations with reflective lookups, which removes the reference from the constant pool. This makes static analysis more difficult, as the actual execution flow is determined at runtime [36]. Reflection is available in many JVM-based programming languages such as Java and Kotlin, although its usage may vary depending on the concrete implementation.

We demonstrate reflective obfuscation with examples from listing 13 and 14. Listing 13 shows the code before reflection, in which an instance of `ExampleClass` is created and its `exampleMethod()` is invoked.

```
1 ExampleClass example = new ExampleClass();
2 example.exampleMethod();
```

Listing 13: Source code before reflection

Dong et al. [46] conducted a large-scale investigation and identified the most frequent pattern for reflective calls in obfuscations. This well-known pattern is shown in listing 14 and enables dynamic access to classes and their methods at runtime by using the Java Reflection API [129].

```
1 String className = decrypt("c29tZSBlbmyX="); // "com.example.ExampleClass"
2 String methodName = decrypt("YW5vdGhlciBm="); // "exampleMethod"
3
4 Object c = Class.forName(className).getDeclaredConstructor().newInstance();
5 Method m = c.getClass().getMethod(methodName);
6 m.invoke(c);
```

Listing 14: Source code after string and reflection-based obfuscation

The pattern typically begins with `Class.forName()`, which loads a `Class` object into memory based on its fully qualified string name. This allows the program to work with classes that were not known at compile time. Next, `Class.getMethod()` retrieves a `Method` object corresponding to a specific public method name and parameter signature. This step allows the inspection and selection of methods. Finally, `Method.invoke()` is called on the selected `Method` object to execute the method on a given instance of the class (or `null` if the method is static). The code obfuscation is further improved by string obfuscation, as otherwise the class and method names would still be visible in plain text.

One disadvantage of reflection is the inherent performance overhead, as dynamic invocations execute significantly slower than direct method calls. In addition, Android 9 (API level 28) implemented a hidden API denylist that restricts access to private or undocumented members via reflection [130]. If an application attempts to use reflection on a denylisted API, the system throws an exception and causes the process to crash. This makes reflection-based obfuscation techniques less effective in newer Android versions and less suitable for long-term obfuscation strategies.

4.4.7 Dynamic Code Loading

Dynamic loading refers to the practice of loading additional code modules or classes into an application at runtime. This means that the actual code is usually retrieved from the file system or an external server. If the DEX file is packaged with the APK, it must be obfuscated (encrypted) to prevent easy extraction and reverse engineering. Unfortunately, dynamic loading with an encrypted DEX file still remains vulnerable to memory dumping and code injection attacks. In Android, dynamic loading is usually achieved with mechanisms such as the `DexClassLoader` [131], `PathClassLoader` [132] or `InMemoryDexClassLoader` [133].

DexClassLoader

The `DexClassLoader` loads classes from packages (`.jar`, `.apk`) that contain a `classes.dex` entry [131]. Listing 15 shows the public constructor of `DexClassLoader`.

```
1 public DexClassLoader (String dexPath, String optimizedDirectory,
2                        String librarySearchPath, ClassLoader parent)
```

Listing 15: DexClassLoader public constructor [131]

The first parameter `dexPath` specifies the file system path to the `.dex` file or package containing the classes to be loaded. The second parameter, `optimizedDirectory` indicates a writable directory to cache optimized classes, but is deprecated since API level 26. The third parameter, `librarySearchPath` is list of directories containing

native libraries, while the parameter `parent` is the parent class loader that serves as a fallback if the `DexClassLoader` cannot find a class.

Listing 16 shows an example of dynamic code loading for obfuscation, however the error handling has been deliberately omitted for demonstration purposes.

```
1  public static void loadCode(Context ctx) {
2      // Path to store the decrypted DEX file
3      File dexFile = new File(ctx.getCodeCacheDir(), "hidden_code.dex");
4
5      // Decrypt the DEX file from assets and save to internal storage
6      decrypt(ctx, "encrypted_hidden_code.dex", dexFile);
7
8      // Define the loader of the decrypted DEX file
9      DexClassLoader dexClassLoader = new DexClassLoader(
10             dexFile.getAbsolutePath(), null, null, ctx.getClassLoader()
11     );
12
13     // Load the class dynamically
14     Class<?> loadedClass = dexClassLoader.loadClass("HiddenClass");
15
16     // Create an instance and invoke the hidden method
17     Object instance = loadedClass.getDeclaredConstructor().newInstance();
18     Method method = loadedClass.getDeclaredMethod("secretMethod");
19     method.invoke(instance);
20 }
```

Listing 16: Dynamic code loading with DexClassLoader

The example method `loadCode(Context ctx)` dynamically loads an encrypted DEX file and execute its method at runtime. The first instruction in line 3 creates a `File` object that represents the location where the decrypted DEX file (`hidden_code.dex`) will be stored and the `ctx.getCodeCacheDir()` call retrieves the absolute path to the application specific cache directory.

In line 6, we define a abstract `decrypt` method that retrieves encrypted DEX file from the assets directory, decrypt it and save it to the previously defined `dexFile`.

The `DexClassLoader` instance is then created using the decrypted DEX file's path, while setting the `optimizedDirectory` and `librarySearchPath` parameters to `null`. In line 14, the `loadClass` method dynamically loads the class `"HiddenClass"` from the decrypted DEX file. After successfully loading the class, an instance of the class is created and the `getDeclaredMethod("secretMethod")` instruction retrieves the method reference from the class. At last, `invoke(instance)` executes the method.

PathClassLoader

The `PathClassLoader` is a simplified version of the `DexClassLoader` and only requires the two parameters `dexPath` and `parent`, as shown in the listing 17 [132].

```
1 public PathClassLoader (String dexPath, ClassLoader parent)
```

Listing 17: PathClassLoader public constructor [132]

The `PathClassLoader` is only able to load classes from the APK itself or the system class path. This restriction makes it unsuitable for scenarios where code needs to be loaded from arbitrary file paths outside of the application.

InMemoryDexClassLoader

The `InMemoryDexClassLoader` can be considered a better alternative to other class loaders for obfuscation, as it loads DEX files directly from memory [133]. The DEX file is never written to the local file system and only resides in a `dexBuffer` (see listing 18), which eliminates the risk of attackers extracting it from the file system. This allows the `InMemoryDexClassLoader` also to bypass the stricter scoped storage policies introduced in Android 10 (API 29+), as it does not require file permissions [83]. A relevant scenario would be downloading a DEX file from an external server and loading it without installing it as part of the application.

```
1 public InMemoryDexClassLoader (ByteBuffer dexBuffer, ClassLoader parent)
```

Listing 18: InMemoryDexClassLoader public constructor [133]

However, using the `InMemoryDexClassLoader` has some disadvantages. First of all, the entire DEX file must be loaded into the memory, which may lead to higher memory consumption for larger code. Second, it does not support native libraries, which makes it unsuitable for apps that require dynamically loaded native code. Third, unlike `DexClassLoader`, the `InMemoryDexClassLoader` is supported only on Android 8 (API 26) and later versions [121].

4.4.8 Virtualization-based Obfuscation

Rolles [47] describes virtualization-based obfuscation as a technique that transforms program code into a custom instruction set. A specialized virtual machine (VM) interprets these instructions at runtime. This technique functions as a form of runtime-based obfuscation, as it alters the standard behavior of the execution environment and shields the logic against static reverse engineering. The VM exclusively interprets and executes the obfuscated code.

The process begins by translating sections of Dalvik bytecode or native ARM instructions into a custom virtual Instruction Set Architecture (ISA). Figure 4.3 illustrates this transformation, where the obfuscator replaces the original code block with virtual bytecode and redirects the entry point to an embedded VM. Developers typically implement this VM as a native library (`.so` file) within the APK to ensure performance and security [32]. The security of the obfuscation relies heavily on the design of this custom ISA.

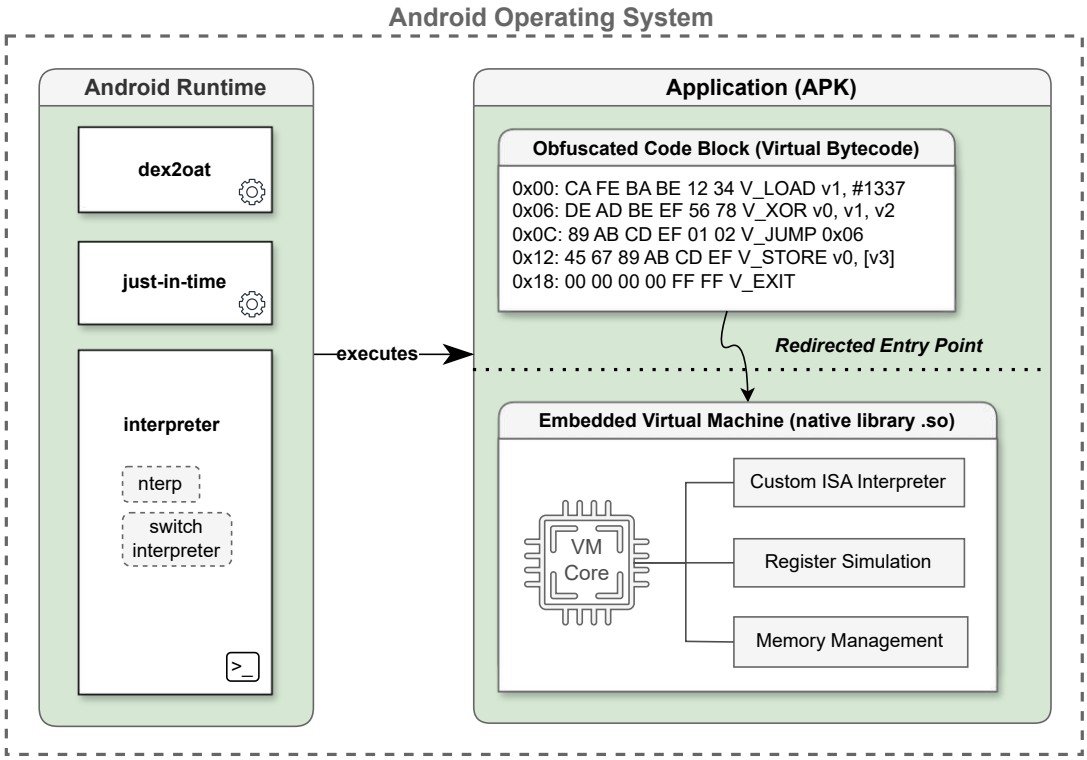

Figure 4.3: Architecture of application-level virtualization-based obfuscation where the VM is embedded as a native library.

The architecture of the embedded VM typically consists of three components:

Custom ISA Interpreter

The core of the virtualization engine is the interpreter, which drives the execution flow. It translates the virtual bytecode into native behavior through two cooperating mechanisms [29]:

- **Dispatcher:** This component functions as the control unit. It runs a continuous loop simulating the "fetch-decode-execute" cycle. The dispatcher fetches the next opcode via the Virtual Program Counter (VPC), decodes the operation, and transfers control to the appropriate handler. Implementations typically rely on large switch statements or lookup tables.
- **Instruction Handlers:** For every unique opcode in the custom ISA, a corresponding handler exists. These are blocks of native machine code that implement the specific logic (e.g., arithmetic, logic operations). Handlers serve as the bridge between the virtual bytecode and the physical CPU.

Register Simulation

The VM encapsulates its entire state within a data structure known as the VM Context. To simulate a physical processor, the VM allocates an array of memory locations that function as Virtual Registers. Additionally, Status Flags mimic a real CPU's status register (e.g., zero flag, carry flag) to store the results of comparisons and arithmetic operations. The VM isolates this virtual state from the host environment. Before the interpreter executes protected code, it saves the host CPU's registers and flags into the context structure. Upon completion, the VM restores this saved state. This context switching ensures that the virtualized code integrates seamlessly with the rest of the application.

Memory Management

The VM manages dedicated memory areas to handle data processing and control flow. A Virtual Stack supports stack-based operations, such as pushing and popping values for function calls or arithmetic computations. Certain designs implement multiple parallel stacks to separate different data types, such as integers and objects. The VPC tracks the location of the next virtual instruction within the bytecode stream.

Security and Performance

Virtualization-based obfuscation is highly effective against static analysis. Disassemblers fail to recognize the virtualized data bytes as valid machine instructions. Consequently, Kinder [48] notes that the original control flow graph remains hidden until a complete semantic understanding of the custom ISA is achieved. Unlike packing or encryption, the process never restores the original code in memory. Modern obfuscators further enhance this by generating a unique, randomized ISA for each application. This variation prevents attackers from applying knowledge gained from one target to another.

However, this security comes at a significant cost. The interpretation of virtual instructions through the fetch-decode-dispatch cycle of a VM introduces a significant performance overhead compared to native execution [28]. This overhead forces developers to limit obfuscation to small, security-critical code sections. Additionally, embedding the VM interpreter increases the overall application size.

Despite these strengths, code virtualization remains vulnerable. Pattern matching identifies the predictable structure of the VM's dispatcher loop, while frequency analysis deduces opcode functions for static ISAs [48]. Attackers also utilize dynamic analysis to trace the interpreter's low-level native instructions, although the high volume of operations renders this process tedious. State-of-the-art deobfuscators move beyond literal reconstruction toward semantic analysis. Coogan et al. [49] employ a semantics-based approach using equational reasoning to analyze behavior without fully reversing the VM.

To counter this, commercial solutions such as CodeVirtualizer [134] and VMProtect [116] integrate anti-debugging mechanisms directly into the interpreter.

CHAPTER 5

Virtualization-based Obfuscation within the Android Runtime

Code virtualization provides high resilience against reverse engineering by decoupling the program code from the standard execution environment and enforcing a custom instruction set architecture. Traditional approaches typically rely on application-level virtualization, where the application contains an embedded interpreter to execute a custom instruction set. We propose a system-level virtualization architecture that utilizes the existing ART as a baseline. This approach eliminates the need to develop and embed a custom virtual machine.

We introduce the Android Opcode Permutation Tool (AOPT), a Java-based obfuscation toolchain that reads arbitrary Android applications (.apk) and modifies their compiled bytecode. The core algorithm implements Instruction Set Randomization (ISR) by transforming the standard Dalvik bytecode into a unique, randomized instruction set through a bijective permutation of all available opcodes. To guarantee high compatibility and prevent runtime errors, we implement semantic subgroups that categorize opcodes based on their format and similarity. The tool also injects a boolean metadata flag into the `AndroidManifest.xml` to identify obfuscated applications. Further, AOPT repackages the modified artifacts into a valid APK and signs the archive with a custom keystore to ensure successful installation.

To execute these obfuscated applications, we create a custom Android operating system that serves as the necessary execution environment. Figure 5.1 illustrates the architectural integration between the modified Android Runtime and the obfuscated application. Our modifications allows the ART to parse the injected metadata flag and dynamically select the appropriate execution path. The new `DexInstructionSet` component manages lookup tables for both original and permutated opcodes. This structure enables a multi Instruction Set Architecture (ISA) environment within the runtime, in contrast

to the single ISA of the standard Android operating system. We specifically extend the `MethodVerifier` to use this component, which ensures that permuted bytecode passes mandatory integrity checks without triggering validation errors. To handle the bytecode execution, we integrate a secondary interpreter derived from the standard `C++` switch mechanism. This component resolves the permuted opcodes and ensures the secure execution of obfuscated applications alongside standard ones. The proposed architecture lays the foundation for future extensions to other runtime mechanisms, such as AOT compilation, JIT compilation, and the `nterp` interpreter.

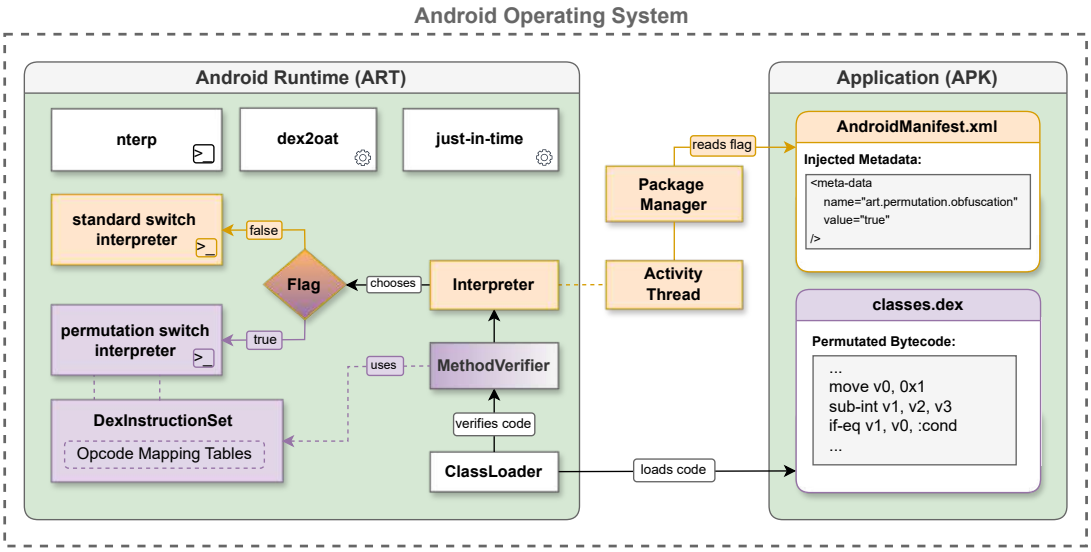

Figure 5.1: Architecture of our system-level virtualization-based obfuscation where we extend the Android Runtime.

5.1 Requirements and Environment

The modified Android operating system is built from Android 15 (Vanilla Ice Cream) and runs on an emulated device with `x86_64` architecture [135]. The build process utilizes the engineering configuration (`eng`) to enable root access and advanced debugging capabilities via the ADB. We provide the detailed commands for the repository setup, build process, and emulator configuration in Appendix B.2.

The host machine runs the Ubuntu 22.04.3 LTS operating system and meets the official hardware requirements for Android platform development [136]. The initial build on Android 13 required only 32 GB of RAM, whereas the transition to Android 15 required a minimum of 64 GB. Appendix B.2 lists the specific hardware specifications of the workstation.

5.2 Analysis of the Android Open Source Project

We analyze the source code of the Android operating system, better known as the Android Open Source Project (AOSP) with branch version `android-15.0.0_r14` [135]. Figure 5.2 presents a subset of the AOSP source tree. The root directory contains the core components that implement the primary operating system services. We briefly describe the high-level directories, as organized in the official source tree [135], before examining the core components relevant to our virtualization-based obfuscation:

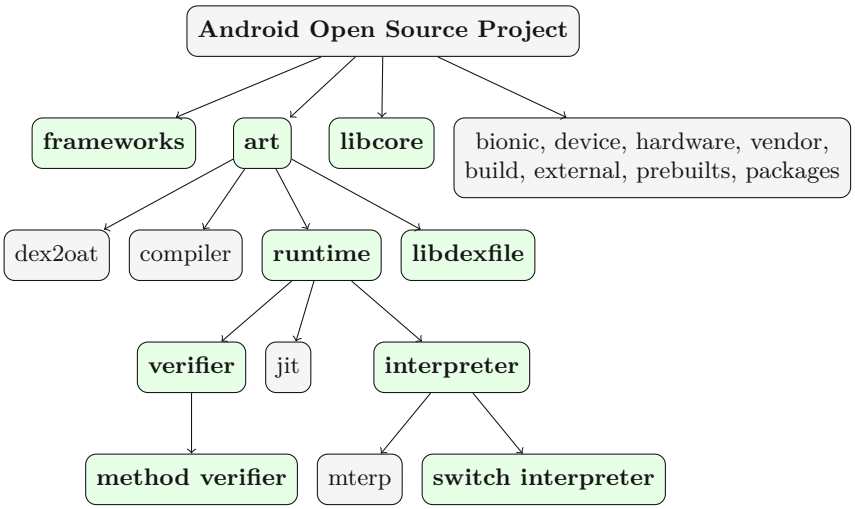

Figure 5.2: Hierarchical structure of important AOSP components. Green nodes highlight the components modified to implement the virtualization-based obfuscation [135].

The `bionic/` directory contains Android's custom C standard library implementation, which Levin [50] characterizes as being specifically optimized for the constraints of mobile devices. The `build/` directory houses the build system rules and configuration files, including the Soong build system definitions. Device-specific configurations and drivers reside in the `device/` directory, while `vendor/` contains proprietary binaries and drivers from hardware manufacturers. The `external/` directory stores open-source libraries from third parties, such as SQLite and Boringssl. The `hardware/` directory defines the HAL interfaces that decouple the Android framework from hardware implementations. The `packages/` directory holds standard Android applications such as Settings, Launcher, and SystemUI. System-level native binaries, such as the init process and logcat, reside in the `system/` directory. Finally, `prebuilts/` contains precompiled binaries and tools required for the build process.

5.2.1 Android Runtime (art/)

The `art/` directory contains the source code for the Android Runtime, which executes Dalvik bytecode. The `dex2oat/` subdirectory orchestrates the Ahead-Of-Time compila-

tion process and leverages profile data from the JIT compiler to optimize hot code paths. It reads the `classes.dex` file and invokes the compiler to generate native machine code, which the system stores as .oat files for execution. The `libdexfile/` directory contains the core libraries for parsing and manipulating the DEX file format. It provides the utilities to load `DexFile` instances, which allows other components such as the verifier and the compiler to access the underlying bytecode structure and instruction data.

The `runtime/` subdirectory contains the core logic of the runtime environment, where we find the `verifier/`, `jit/` and `interpreter/` components. The verifier guarantees the structural integrity and type safety of the bytecode prior to execution. The `jit/` directory contains the Just-In-Time implementation.

The `interpreter/` component contains bytecode interpreter for debugging, fallback execution, and class initialization. This includes both the high-performance assembly interpreter (`mterp`) and the portable C++ switch-based interpreter implementation. The `mterp/` directory historically houses the *Modular Interpreter*, which relies on hand-optimized assembly fragments for each opcode to achieve high performance. The files `nterp.cc` and `nterp.h` implement the *New Interpreter*, a modern replacement introduced in Android 11 [137]. Although technically distinct, `nterp` resides in this directory because it succeeds `mterp` as the primary fast interpreter. Unlike its predecessor, `nterp` aligns its stack frame layout with compiled code, which allows for efficient, zero-overhead transitions between interpreted and native execution [137].

5.2.2 Android Frameworks (frameworks/)

The `frameworks/` directory provides the essential Java APIs and services that Android applications utilize. It serves as an abstraction layer between the application layer and the underlying native system services. The `frameworks/base` directory is especially relevant for our obfuscation, since it contains the core system services and application lifecycle management logic. From an architectural perspective, this component orchestrates the configuration of the execution environment based on the application manifest. For example, it includes critical components such as the `ActivityManager` and `PackageManager`. It also defines the `Binder` IPC mechanism that enables communication between applications and system services.

5.2.3 Core Libraries (libcore/)

The `libcore/` directory provides the Java class libraries that form the basis of the Java Runtime Environment (JRE) on Android. While `frameworks/` provides Android-specific APIs, `libcore/` implements the standard OpenJDK libraries. This directory bridges the gap between the Java language layer and the native system capabilities of `bionic/` and the Linux kernel. Applications rely on these libraries for file I/O, networking, and data structures. Modifications to the runtime environment often require corresponding adjustments in `libcore/` to ensure that core native methods interact correctly with the modified virtual machine execution state.

The examination of `libcore/` concludes our analysis of the AOSP. The directory `frameworks/base/` controls application initialization, while `runtime/interpreter/` contains the engine for bytecode execution. We leverage this architectural knowledge to integrate our obfuscation mechanism directly into the execution environment while preserving the integrity of the core libraries. Before we describe the technical modifications of the runtime, we define our code virtualization approach. The following section presents the permutation algorithm that provides a unique instruction set for execution.

5.3 Instruction Set Randomization

Kc et al. [51] describe Instruction Set Randomization (ISR) as a technique to protect software by transforming the standard instruction set into a unique, randomized version. Only a specific, modified execution environment is able to interpret this randomized ISA. Our implementation achieves ISR by applying opcode permutation to the existing Dalvik bytecode within the Android Runtime. This approach functions as a lightweight form of code virtualization and eliminates the overhead of developing a custom virtual machine.

We generate the randomized instruction set by shuffling the available Dalvik opcodes via a randomization seed. Algorithm 5.1 defines the generation of this bijective mapping, which serves as the foundation for both encoding the application and decoding instructions at runtime.

Algorithm 5.1: Bidirectional Opcode Permutation Mapping

Input: A list of opcodes L, a seed number s
Output: A mapping $M : \text{Opcode} \rightarrow \text{Opcode}$

```
1  M ← ∅;
2  L' ← Fisher-Yates Shuffle(L, s);
3  for i ← 0 to |L'| − 2 step 2 do
4  |  M(L'[i]) ← L'[i + 1];
5  |  M(L'[i + 1]) ← L'[i];
6  end
7  if |L'| is odd then
8  |  M(L'[last]) ← L'[last];
9  end
10 return M;
```

The algorithm begins by shuffling the input list of opcodes L using the seed s, which produces a randomized list L'. To ensure a uniform distribution where every permutation is equally likely, we employ the *Fisher–Yates shuffle* as described by Knuth [52]. At line 3, the algorithm then iterates over the randomized list L' in steps of two and creates bidirectional pairs by mapping each opcode to its adjacent one and vice versa. If the total number of opcodes is odd, the last unpaired opcode is mapped onto itself to ensure a valid assignment. However, this means that no obfuscation is applied to this opcode. Finally, the last line simply returns the permutation mapping M.

The algorithm has a runtime complexity of $O(n)$, since the shuffle and the pairing loop each require a single pass through the list.

5.3.1 Search Space and Cryptographic Strength

The Android Dalvik bytecode format defines a fixed range of 256 possible opcodes [138]. We establish this value as the theoretical upper bound for the permutation space. In practice, the number of active opcodes varies depending on the specific Android or ART version. The number of bijective mappings for a set of n opcodes corresponds to $n!$ (factorial). Consequently, a full set of 256 opcodes results in 256! potential combinations, which is approximately 8.57×10^{506}.

This value represents an astronomical increase in complexity compared to modern cryptographic standards. For example, AES-256 possesses a key space of 2^{256}, or approximately 1.16×10^{77}. The permutation space of 256! exceeds the AES-256 key space by a factor of roughly 10^{429}. Even if technical constraints limit the permutation to 100 opcodes, the resulting $100! \approx 9.33 \times 10^{157}$ combinations still outperform the brute-force resistance of 128-bit symmetric keys. This search space ensures that every protected application instance utilizes a unique randomized instruction set architecture. Attackers cannot reuse mapping data from one application to deobfuscate another because instruction semantics remain instance-specific for each device.

However, this theoretical complexity assumes that an attacker relies solely on brute-force attempts. In practice, automated analysis techniques significantly reduce the effective search space. Attackers employ frequency analysis to compare the occurrence of virtual opcodes against known statistical distributions of standard Dalvik instructions. Furthermore, control flow analysis poses a significant risk. Even when opcode values change, the underlying semantics of the instruction remain visible. Examples of these semantics include the number of operands or the effect on the program counter. By observing the side effects of an instruction, such as memory access patterns or register modifications, an analyst infers the original instruction type. Since the permutation table remains static throughout the application execution, recovering a small subset of opcodes provides enough context to deduce the remaining program logic.

5.4 Bytecode Permutation of Android Applications

The Android Opcode Permutation Tool (AOPT) implements a pipeline that transforms standard Dalvik bytecode into a permuted instruction set architecture for our virtualization-based obfuscation. Figure 5.3 presents the architectural stages of this pipeline, which the following subsections describe in detail.

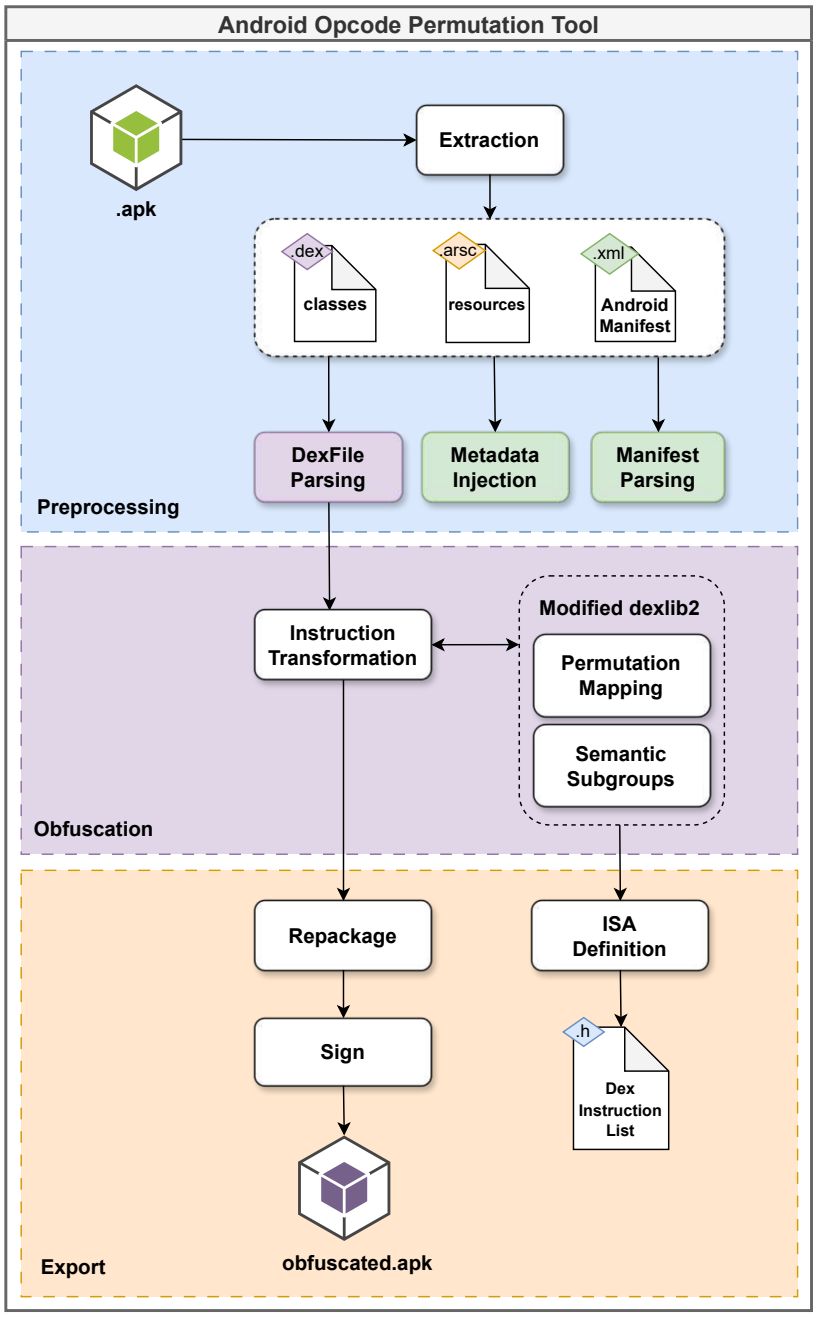

Figure 5.3: Obfuscation pipeline of AOPT

5.4.1 Preprocessing

The pipeline begins with the *Extraction* phase. AOPT utilizes `apktool` to unpack the target APK file and extracts the `AndroidManifest.xml`, `classes.dex` files, and `resources.arsc` into a working directory. During *Manifest Parsing*, the tool identifies the target API version to select the corresponding opcode set, or else defaults to api level 34 (current highest supported api level). Next, AOPT performs *Metadata Injection* by inserting a boolean flag with the sample key `art.permutation.obfuscation` and the value `true` into the manifest. This flag serves as a control flow identifier for our modified ART. To conclude the preprocessing phase, AOPT performs *DexFile Parsing* by loading the .dex files into Java `DexFile` objects provided by `dexlib2`.

5.4.2 Obfuscation

The core obfuscation logic happens in the *Instruction Transformation* process.

Our implementation requires two important modifications to the `dexlib2` library. First, we disable the internal opcode format validation. Standard library checks prevent the serialization of instructions where the opcode does not match the expected register format, which occurs frequently during random permutation. Second, we create a new `InstructionToBuilderInstructionConverter` class. This component supports the modification of existing instructions by converting them into a builder format while applying the opcode swap. This conversion ensures that the resulting Dalvik bytecode maintains structural integrity for the reassembly process despite the semantic corruption.

Section 5.3 already establishes the baseline algorithm that AOPT uses to create a bidirectional, one-to-one mapping between Dalvik opcodes. The algorithm uses a seeded random generator to ensure that the same input seed consistently produces the same ISA for both the app and the custom runtime.

The implemented algorithm follows a tiered grouping approach to ensure the functional integrity of the transformed application:

- **API-Level Filtering:** The tool first identifies all valid opcodes for the target Android API level and filters out deprecated instructions, such as `QUICK` opcodes or specific return barriers.
- **Semantic Subgroups:** AOPT prioritizes critical opcodes by processing them within predefined semantic groups, such as `INVOKE` or `FIELD_ACCESS`. This ensures that instructions with sensitive or specialized behaviors only swap with opcodes from the same functional category.
- **Format-Based Alignment:** For all remaining instructions, the algorithm groups opcodes by their structural format. This step ensures that an instruction is only swapped with another that shares the same register requirements and bit-length, which prevents layout corruption in the DEX file.

- **Bidirectional Pairing:** Within each identified group, the tool shuffles the list and pairs opcodes sequentially. If opcode A maps to opcode B, then opcode B automatically maps back to opcode A. In cases where a group contains an odd number of instructions, the final opcode maps to itself to maintain a complete set.

As a result of this workflow, any opcode not meeting these criteria is mapped to itself, which ensures the tool remains robust even when encountering unsupported instructions.

5.4.3 Export

The final *Export* phase generates and exports the software artifacts.

The *ISA Definition* process bridges the gap between the application and the runtime. It generates a `C++` header file (`dex_instruction_list.h`), which is the permutated instruction set definition for the ART. This file acts as the symmetric obfuscation key and is required to compile the custom interpreter. For traceability and debugging purposes, AOPT also produces a `permutationMapping` file that lists the bijective opcode permutation mapping.

The final stages involve *Repackaging* and *Signing*. The tool bundles the modified DEX files and resources back into an APK container using `apktool` and applies `zipalign` to optimize data alignment. Finally, AOPT generates a dummy keystore and signs the APK using the `apksigner` utility. The process concludes with the storage of the final artifacts on the local file system.

While AOPT successfully produces an APK with permuted bytecode and embedded metadata, standard Android devices cannot execute this format. The following section details the necessary modifications to the Android Runtime to interpret this custom instruction set.

5.5 Modification of the Android Runtime

We create a custom Android OS to execute applications protected by our opcode permutation technique. To do that, we extend the AOSP architecture to recognize obfuscated applications via the custom metadata flag and execute them with their permuted instruction set. We restrict our modifications exclusively to the switch interpreter and do not implement the permutation logic within nterp, JIT, or AOT compilation paths. Consequently, the Android operating system must use only the fallback switch interpreter to test the obfuscation. Standard Android systems default to nterp (assembly-based interpreter) or AOT-compiled machine code. In addition to the emulator flags, we modify the `CanRuntimeUseNterp()` method within the `nterp.cc` file to always return `false`. In our experience, this is necessary to force the Android operating system to use the fallback switch interpreter completely and not to switch to the assembly interpreter in certain cases.

This following sections describes the implementation of the metadata parsing, the abstraction of the instruction set, and the obfuscated interpreter.

5.5.1 Metadata Flag Parsing

We extend the metadata parsing architecture to extract the obfuscation flag from the application package during installation. The custom ART uses this flag to identify obfuscated applications. We modify the `frameworks/base` module and target the `Package Manager` for manifest parsing. The `ActivityThread` handles the application startup. We also extend the `libcore` and `art` modules to establish a JNI bridge. This bridge propagates the obfuscation flag from the Java framework to the native runtime environment.

Package Parsing

We define the new interface methods `setPermutationObfuscation(boolean)` and `isPermutationObfuscation()` in `ParsingPackage`. This builder interface enables the parsing logic to set properties on the mutable package object. We add the metadata key `art.permutation.obfuscation` within `ParsingPackageUtils`. The `setPermutationObfuscationFlag` method extracts this boolean value from the `<meta-data>` bundle of the application during package parsing. The parser updates the internal package representation if the flag evaluates to true. We modify `PackageImpl` to store this state in the boolean field `mPermutationObfuscation`. The `isPermutationObfuscation` method exposes the field. We also declare the method in the `AndroidPackage` interface. This exposes the property to system components requiring a read-only package view and ensures the persistence of the obfuscation status.

Runtime Propagation

We map the internal package state to a flag in `ApplicationInfo` to make the status available to the application process at runtime. We introduce the private extension flag `PRIVATE_FLAG_EXT_PERMUTATION_OBFUSCATION`. The `AppInfoUtils` class sets this bit in `privateFlagsExt` during the generation of the `ApplicationInfo` object. This flag enables efficient identification of obfuscated apps without reparsing the manifest.

The `ActivityThread` executes the final propagation step during application binding. We retrieve the flag from the `ApplicationInfo` object within `handleBind Application` before the application code executes. The thread passes this boolean value to the internal API class `dalvik.system.RuntimeHooks`. We add the static method `setPermutationObfuscation(boolean)` to `RuntimeHooks` and register it in `libcore/api/module-lib-current.txt` to satisfy module API requirements.

Finally, we implement the native hook counterpart in the newly created `art/runtime /native/dalvik_system_RuntimeHooks.cc` file and its associated header. We

register the source file in the build configuration `art/runtime/Android.bp`. The JNI function propagates the state to the global `Runtime` instance via `runtime->SetPermutationObfuscationEnabled()`.

These modifications ensure that the ART instance receives the obfuscation flag before interpreter initialization and guarantees the correct execution mode.

5.5.2 Abstraction of the Instruction Set

The standard Android Runtime uses global, static arrays to store instruction properties such as names, formats, and flags. This design prevents the simultaneous execution of standard and obfuscated bytecode within the same process. We introduce the `DexInstructionSet` class to encapsulate these properties into an instantiable object. This class manages vectors for instruction names, descriptors, and a reverse opcode map. The map translates raw byte values back to internal opcode constants. We place the class in `art/libdexfile/dex/` and register it in the `art/libdexfile/Android.bp` build configuration.

Preprocessor macros initialize these vectors in the `dex_instruction.cc` file. The original `dex_instruction_list.h` file defines the standard bytecode mapping. We introduce the parallel header `dex_instruction_list_obfuscation.h` to define the permuted opcode values. This file represents the symmetric key generated by the AOPT obfuscator (Section 5.4.3). The `DexInstructionSet` implementation provides the global accessor functions `GetStandardDexInstructionSet` and `GetObfuscation DexInstructionSet`. The `ClassLoaderContext` retrieves the global obfuscation status from the Runtime and passes it to the `DexFileLoader` during APK loading. This associates the resulting `DexFile` objects with the correct instruction set instance.

We overload the accessor functions of the `Instruction` class to accept a reference to the `DexInstructionSet`. These context-aware methods map raw byte values to internal opcode constants via the provided instruction set. Runtime components, such as the `MethodVerifier`, pass the active instruction set instance to these methods to interpret the obfuscated stream correctly. We integrate `DexInstructionSet` into the `SafeDexInstructionIterator` to calculate instruction boundaries during iteration. For example, the overloaded `VRegB_4rcc` method uses the `DexInstructionSet` object to validate the instruction format before register extraction. We implement the original parameterless functions as wrappers. These wrappers default to the standard instruction set and preserve backward compatibility with the codebase.

5.5.3 Obfuscated Switch Interpreter

We implement the virtualization-based obfuscation within the fallback switch interpreter of the Android Runtime. This `C++` interpreter serves as a reference implementation and handles execution when the runtime bypasses the optimized assembly interpreters. Figure 5.4 depicts the file hierarchy of our modified ART interpreter.

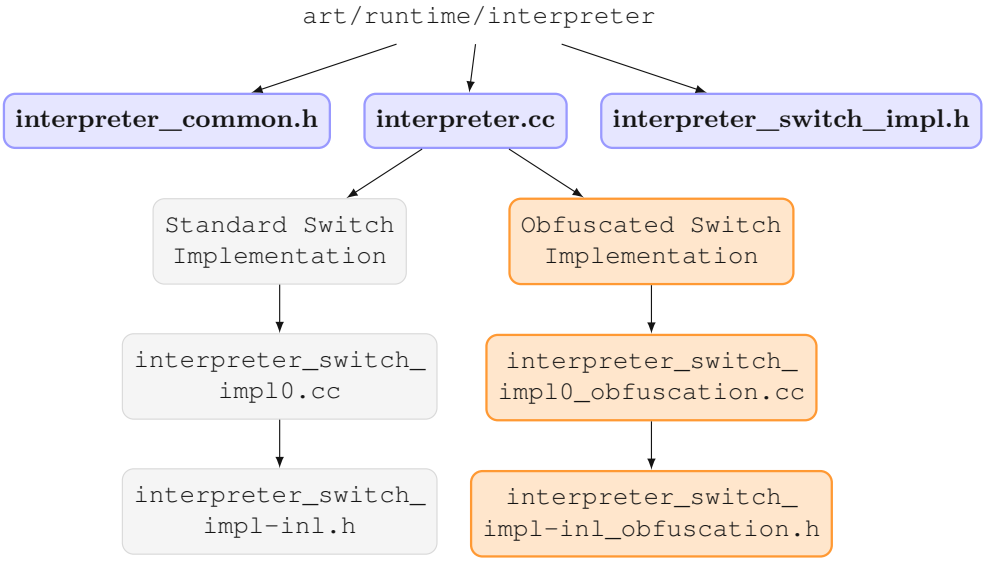

Figure 5.4: File hierarchy of the modified ART interpreter. Bold blue nodes are modified core components, grey nodes show the standard implementation, and orange nodes the new obfuscation files.

The `ExecuteSwitch` function in `interpreter.cc` acts as the primary entry point. It retrieves the `DexFile` associated with the current `ShadowFrame` and verifies the obfuscation status. The interpreter selects the obfuscated switch implementation if the `is_obfuscated` flag evaluates to true. Otherwise, it defaults to the standard switch implementation. This mechanism ensures that standard applications use the unmodified interpreter, while protected applications use the permuted instruction set.

The `interpreter_switch_impl.h` header is the essential binding interface that propagates the instruction set configuration to the execution logic. We modify the `SwitchImplContext` structure to include a reference to the `DexInstructionSet`. This addition allows the dispatcher to pass the specific opcode mapping table directly to the implementation layer. The `ExecuteSwitchImpl` wrapper uses this context to initialize the runtime state before it invokes the assembly bridge. Consequently, the core interpreter loop accesses the correct permutation data via this context to ensure accurate instruction decoding for both standard and obfuscated control flows.

We also extend `interpreter_common.h` to introduce polymorphic function templates that accept the `DexInstructionSet` as a parameter. These templates enable core operations such as method invocation (`DoInvoke`), branching (`DoPackedSwitch`), and array creation (`DoFilledNewArray`) to decode operands correctly regardless of the underlying opcode permutation.

Instruction Dispatch

The `interpreter_switch_impl0_obfuscation.cc` source file acts as the compilation unit. It instantiates the templates defined in the header file and compiles the logic into executable machine code. The inline header file, `interpreter_switch_impl-inl_obfuscation.h`, contains the core execution logic. This file is a duplicate of the standard `interpreter_switch_impl-inl.h`, but uses the permuted instruction set. Listing 19 presents the `ExecuteSwitchImplObfuscationCpp` function, which handles instruction execution. For presentation purposes, we reduced the code to the essentials in order to focus on the instruction dispatch logic.

```
template<bool transaction_active>
void ExecuteSwitchImplObfuscationCpp(SwitchImplContext* ctx) {
  // Retrieve the dex instruction set from context
  const DexInstructionSet& dex_instruction_set = ctx->dex_instruction_set;

  while (true) {
    const Instruction* const inst = next;
    // Fetch the raw 16-bit instruction data (opcode + other data)
    uint16_t inst_data = inst->Fetch16(0);
    // Exception check based on real opcode
    DCHECK_EQ(self->IsExceptionPending(),
    inst->Opcode(inst_data, dex_instruction_set) ==
    Instruction::MOVE_EXCEPTION);

    // Switch on the lower 8 bits, which represent the opcode value
    switch (inst_data & 0xff) {
#define OPCODE_CASE(OPCODE, OPCODE_NAME, NAME, FORMAT, ...)       \
    case OPCODE: {                                                \
        /* Calculate the next program counter */                  \
        next = inst->RelativeAt(                                  \
           Instruction::SizeInCodeUnits(Instruction::FORMAT));    \
        /* Execute the C++ handler for this opcode */             \
        bool success = OP_##OPCODE_NAME<transaction_active>(      \
            ctx, instrumentation, self, shadow_frame, dex_pc,     \
            inst, inst_data, next, exit);                         \
        if (success) { continue; }                                \
        break;                                                    \
      }
      // Expand the case labels using the obfuscated instruction list
      DEX_INSTRUCTION_LIST_OBFUSCATION(OPCODE_CASE)
#undef OPCODE_CASE
    }
    // ... (Error handling and exit logic)
  }
}
```

Listing 19: Macro-based generation of switch cases mapping permuted opcodes to instruction handlers inside the `ExecuteSwitchImplObfuscationCpp` function.

The function first retrieves the dex_instruction_set from the context parameter. It then executes a continuous while loop that fetches the 16-bit instruction data (inst_data) from the current program counter. Line 11 demonstrates the necessity of the dex_instruction_set with the DCHECK_EQ macro. The macro recognizes a pending exception if and only if the current instruction executes a MOVE_EXCEPTION operation. We pass the dex_instruction_set to the Opcode method because the raw inst_data contains only the permuted opcode value. The Opcode method uses the specific mapping table stored within dex_instruction_set to resolve the obfuscated value back to its original instruction identity.

The term "switch interpreter" derives from the central C++ switch construct that handles the instruction dispatch in line 16. The statement inst_data & 0xff extracts the opcode by performing a bitwise AND operation on the 16-bit code unit from memory. The interpreter uses the DEX_INSTRUCTION_LIST_OBFUSCATION macro to generate the switch cases via the X-Macro pattern. A local OPCODE_CASE definition captures the OPCODE, OPCODE_NAME, and FORMAT arguments. The macro uses these arguments to map each permuted opcode to its handler. Consequently, the code decodes operands for the permuted set without requiring runtime format resolution.

The need for a distinct header file and execution method comes from the static nature of the C++ switch construct. Case labels within a switch statement must be compile-time constant expressions. The compilation process therefore fixes the mapping between a raw opcode value and its corresponding instruction handler. While the DexInstructionSet within the runtime context provides dynamic information regarding instruction formats and flags, it cannot alter the hardwired dispatch logic of the switch statement.

The standard interpreter uses the DEX_INSTRUCTION_LIST macro to generate case labels that match the default Android bytecode specification. In contrast, interpreter_switch_impl-inl_obfuscation.h employs the DEX_INSTRUCTION_LIST_OBFUSCATION macro. Separate compilation of these methods generates two independent optimization jump tables. This routing ensures that the interpreter directs permuted opcodes to their respective implementation logic without additional runtime overhead.

This integration concludes the technical implementation of the virtualization-based obfuscation. The system establishes a parallel execution environment that interprets permuted bytecode while maintaining full compatibility with the Android Runtime's internal logic. The next chapter evaluates the effectiveness of this obfuscation technique against reverse engineering and measures the performance impact of the modified runtime environment.

CHAPTER 6

Evaluation of Virtualization-based Obfuscation

This chapter evaluates the correctness, performance and security of the proposed virtualization-based obfuscation. Section 6.1 describes the experimental environment and the technical configuration of the custom Android operating system. Section 6.2 introduces the test application suite, which includes the 20 applications used for benchmarking. We define the qualitative and quantitative evaluation metrics in Section 6.3. Section 6.4 presents the empirical results. Finally, Section 6.5 provides a discussion of the findings and their implications for Android application protection.

6.1 Experimental Environment

The evaluation was conducted using the same host and AOSP environment described in Chapter 5. The standard and modified Android OS (Android 15.0.0_r14 [135]) ran on an x86_64 emulator instance [110]. To isolate the performance impact of the obfuscation, the virtual device was configured with hardware-accelerated graphics (Host GPU) and utilized the standard AOSP engineering build resource allocation (approx. 2 GB RAM). We utilized the `eng` build variant to enable root access and advanced debugging via `adb`, which was necessary for extracting precise memory and timing logs.

Decompilation was performed using the *JADX v1.5.3* [114] tool, while the CFG analysis was performed using *Ghidra v12.0* [115]. Multiple metrics and experiments were captured using various `adb` shell commands and the `dumpsys` diagnostic tool.

6.2 Test Applications

To evaluate the robustness and compatibility of our opcode permutation obfuscation technique, we selected a diverse set of 20 Android applications. Table 6.1 shows the full list of our chosen test applications with its concrete versions, categories and a complexity classification.

Application	Ver.	Category	Complexity	Ref.
Algorithmic Benchmark	1.0.0	Tools	Low	-
Chess	9.9.15	Games	Medium	[139]
Feeder	2.16.1	News	Medium	[140]
Fossify Calculator Beta	1.3.0	Tools	Low	[141]
Fossify Calendar	1.9.0	Productivity	Medium	[142]
Fossify Clock Beta	1.5.0	Tools	Low	[143]
Fossify Contacts	1.5.0	Comm.	Medium	[144]
Fossify File Manager	1.5.0	Tools	Medium	[145]
Fossify Gallery	1.10.0	Photography	Medium	[146]
Fossify Home	1.5.0	Personal.	High	[147]
Fossify Messages	1.7.0	Comm.	Medium	[148]
Fossify Music Player	1.6.0	Music & Audio	Medium	[149]
Fossify Notes	1.6.0	Productivity	Medium	[150]
Fossify Paint	1.2.0	Graphics	Low	[151]
FREE Browser	3.4	Comm.	High	[152]
OpenMoneyBox	3.5.1	Finance	Medium	[153]
QuickWeather	2.8.5	Weather	Medium	[154]
Simple Clipboard Editor	1.5	Tools	Low	[155]
Sudoku Privacy	3.2.5	Games	Low	[156]
TicTacToe Game	1.0.0	Games	Low	[157]

Table 6.1: Suite of Android applications for the evaluation.

The *Algorithmic Benchmark* application is our self developed, base application used for prototype validation, performance testing and for qualitative analysis. The remaining 19 applications are from the F-Droid repository [158]. This selection criteria prioritizes open-source availability to ensure reproducibility and analysis of the source code if execution failures occur.

We categorize the test applications into three complexity tiers:

1. **Low Complexity:** Standalone applications with minimal external dependencies. These apps primarily test basic Dalvik bytecode execution and standard UI widget rendering.

2. **Medium Complexity:** Applications that are likely to involve local database operations (e.g. SQLite), background services, and file I/O. These test the stability of the obfuscated runtime during data persistence and inter-component communication.

3. **High Complexity:** Apps requiring significant hardware integration, network usage, or complex rendering pipelines. For example, a browser relies heavily on the Android System WebView and tests the interoperability between our permutated opcode interpreter and native system components.

The chosen applications range from simple utilities to complex, feature-rich tools to test whether our modified ART is capable of handling various architectural patterns and resource demands.

6.3 Evaluation Metrics

This section details the specific metrics used to assess the impact and effectiveness of the opcode permutation obfuscation. We categorize the assessment into three primary dimensions: functional correctness, performance, and resilience against reverse engineering.

6.3.1 Functional Correctness & Stability

The obfuscated application must execute without errors and exhibit the exact same behavior as the original version. We assess this correctness by manually installing and launching each application on our custom Android Runtime. We interact with the user interface and trigger core features to verify that the app operates as intended. We evaluate correctness through the following metrics:

Compatibility Rate
We classify each application into one of three states based on the observations collected during the manual execution:

- *Fully Functional (✓)* : The app launches and core features work without error.
- *Partially Functional (≈):* The app launches but exhibits minor UI glitches or non-critical errors.
- *Broken (×) :* The app crashes immediately or fails to launch.

Stress Testing
We evaluate the robustness of the obfuscation using the UI/Application Exerciser Monkey [159]. This is achieved by interfacing with the emulator via ADB and use the shell command "`adb shell monkey -p <package_name> -v 5000`". The tool generates 5,000 pseudo-random user events, including clicks, touches, and gestures, for each application. In addition, we pin the apps before each stress tests to prevent UI interactions outside of the application. This allows us to compare the execution stability, memory footprint, and user interface responsiveness of original and obfuscated apps.

6.3.2 Performance

The obfuscated application must maintain acceptable execution speeds and storage efficiency to preserve usability. We assess the operational overhead introduced by the modified Android Runtime and compare artifact sizes of original applications against their obfuscated counterparts. We quantify the performance cost using the following metrics:

Launch Latency
We measure the cold start time using the command `adb shell am start -W -n <activity>`. The metric is the `TotalTime` value, which represents the duration from the intent initiation until the first frame is drawn. We compare the mean time of 5 runs between the original and obfuscated applications.

Execution Throughput
Algorithmic tasks allow us to measure the performance without the noise of UI rendering or I/O latency, and provides a precise metric for the efficiency of the main execution loop. To measure pure algorithmic overhead, we utilize standard micro-benchmarks that perform heavy integer and floating-point calculations. While launch time measures the overhead of initialization and class loading, it does not capture the continuous effort by the CPU during sustained execution. Since every instruction in the obfuscated app must be intercepted and mapped by our modified ART before execution, we hypothesize a reduction in raw processing speed.

We found no suitable performance app that provided such simple, CPU bound benchmarks. The only real candidate was the *PassMark PerformanceTest* [160] application, but it was not usable with our obfuscation due to built-in integrity checks. That is the reason, why we developed the *Algorithmic Benchmark* app ourselves and provided three benchmarks: Fibonacci Recursive, Fibonacci Iterative and BubbleSort. To keep it simple, we track the seconds to complete each benchmark and average it across 5 runs.

Disk Space
We calculate the percentage increase in file size for both the final APK package and the internal .dex files. The APK file format acts as a container (ZIP archive) for various assets, resources, and signatures that should be unaffected by our obfuscation. Since our technique operates exclusively on the Dalvik bytecode, the .dex files are the only components where structural changes occur.

6.3.3 Security & Resilience

The primary goal of a obfuscation technique is to hinder reverse engineering. Therefore, we measure the strength of its protections using the following indicators:

Decompilation Errors & Method Success Rate
To quantify the resilience of the obfuscated artifacts against reverse engineering, we utilize the *Total Errors* and *Method Success Rate* reported by the JADX decompiler.

It is important to distinguish between *decompilation success* and *semantic integrity*. The success rate metric solely indicates the decompiler's ability to reconstruct a syntactically valid Abstract Syntax Tree (AST) and export it as Java source code. It does not validate the logical equivalence of the output against the original program.

In the context of opcode permutation, a high success rate is deceptive. For instance, replacing an `ADD_INT` opcode with `SUB_INT` results in valid Java syntax. JADX classifies this as a successful decompilation event, because the resulting code complies with Java grammar rules. However, the semantic logic is fundamentally compromised. Consequently, a high success rate does not represent a failure of the obfuscation, but rather quantifies the volume of deceptive code that appear correct to a reverse engineer but conceal the true runtime behavior executed by the custom interpreter.

Readability & Compatibility with Traditional Obfuscations
We conduct a qualitative analysis of the decompilation output from JADX to assess readability before and after applying our obfuscation. In addition, we compare the compatibility and impact of using traditional obfuscation techniques (e.g. identifier renaming and junk code insertion) in combination with our opcode permutation technique. For the traditional obfuscation techniques, we apply the *ArithmeticBranch*, *FieldRename*, *MethodRename* techniques from the *Obfuscapk* tool [161]. We limit the qualitative analysis to one application (*Algorithmic Benchmark*) in order to stay within the scope of this thesis.

Control Flow Graph Distortion
We compare the structural integrity of the CFG between the original and the obfuscated application. A successful obfuscation influences the CFG and makes it disjointed or more complex compared to the original application. We utilize Ghidra [115] (Function Graph View) to reconstruct the CFG of the `runBenchmarks` method within the *Algorithmic Benchmark* application.

Dynamic Instrumentation Resistance
We test the ability of dynamic instrumentation frameworks (e.g. Frida) to hook method calls at runtime. Success is measured by the immediate termination (crash) of the application upon hook injection, which indicates that the modified runtime correctly rejected the foreign standard bytecode.

6.4 Results

This section presents the results of our evaluation across three primary dimensions. First, we establish the functional reliability of the system by examining compatibility rates and stress testing results. We then quantify the performance overhead through measurements of execution throughput, launch latency, and storage requirements. Finally, the analysis concludes with an assessment of security resilience, where we investigate decompilation failures, control flow graph distortion, and the effectiveness of the scheme against dynamic instrumentation and traditional reverse engineering tools.

6.4.1 Compatibility Rate

Table 6.2 shows the results of the compatibility analysis. We achieve a overall success rate of 85%, with 17 out of 20 obfuscated applications executing correctly. We aggregate the 11 applications of the *Fossify Suite* into a single table entry, as they exhibit identical runtime behavior. The successful applications, including our own *Algorithmic Benchmark*, demonstrate full functional equivalence with their original versions.

Application	Status	Remarks
Algorithmic Benchmark	✓	
Chess	✓	
Feeder	✓	
Fossify Suite (11 apps)	✓	
FREE Browser	×	*Register Bounds Violation*
OpenMoneyBox	✓	
QuickWeather	×	*Register Bounds Violation*
Simple Clipboard Editor	✓	
Sudoku Privacy Friendly	✓	
TicTacToe Game	×	*Resource Loading Failure*

Table 6.2: Compatibility analysis of 20 obfuscated applications

Three applications fail to execute and terminate immediately upon launch. The *TicTacToe Game* crashes with a `MissingResourceException` within the `libGDX` game engine framework, specifically indicating a failure to load the `i18n/TicTacToe` bundle. The error trace points to a disruption in the resource resolution pipeline. This failure correlates with potential defects in the repackaging process regarding the preservation of non-code assets, or alternatively, a bytecode execution anomaly that corrupts the class loader context required for internal resource localization.

Free Browser and *QuickWeather* terminate immediately with a `SIGABRT` signal and the specific runtime error "`Check failed: i < NumberOfVRegs()`". This error explicitly confirms a register bounds violation where the interpreter attempts to access non-existent registers. The failure signature is consistent with a gap in the interception logic, where the runtime executes an instruction's literal definition rather than the

intended permuted variant. This bypass violates the register layout defined for the original method. Furthermore, the error indicates structural constraint violations within the obfuscation mapping, specifically where bidirectional swaps between opcodes of incompatible register widths trigger boundary checks during static stack verification.

The failures of those applications are similar edge cases, which involve native code boundaries (JNI) or complex static initialization sequences that expose structural inconsistencies or missing handler implementations in the custom runtime.

6.4.2 Stress Testing

Most obfuscated applications survive the full stress test without termination. However, three applications crash during the test execution: *Fossify Paint*, *FREE Browser*, and *OpenMoneyBox*. These crashes coincide with periods of high-frequency input injection, which correlates the instability with the execution overhead of the custom interpreter. The data indicates that the additional translation latency prevents these event-heavy applications from processing input within the system's required time constraints. Furthermore, the exception logs collected during these failures point to specific edge cases where the opcode permutation logic encounters unmapped instruction sequences in complex control flows.

Figure 6.1 shows the percentage of Janky frames, which represent dropped or delayed frames during UI rendering. The original applications exhibit an average Janky frame rate of 20.02%, whereas the obfuscated applications average 21.30%. We attribute the instances where obfuscated apps outperform their original counterparts to the non-deterministic nature of the `Monkey` testing tool. Since the random input generation creates distinct execution paths for each run, the obfuscated trials occasionally traverse less complex UI hierarchies. The *Chess* application acts as a performance outlier, where the janky frame rate increases from 31.37% to 65.98% following obfuscation. Nonetheless, the negligible difference of 1.28% between the average percentages demonstrates that the opcode permutation does not significantly degrade the graphical performance or responsiveness of the applications.

Figure 6.2 compares the total RAM usage after the stress test between the original and obfuscated applications. The original applications utilize an average of 51.75 MB, while the obfuscated versions consume an average of 54.73 MB. This represents a marginal increase of approximately 5.7% across the entire dataset. However, this average is heavily skewed by the *Feeder* application, which exhibits a 75.88% increase in memory usage. For the majority of other applications, the memory footprint remains stable or decreases slightly. We hypothesize that the execution overhead of the obfuscation acts as a throttle during the stress test and limits the rate at which the application accumulates cached resources. Consequently, the obfuscated version exhibits a lower peak memory usage because it traverses fewer resource-intensive states than the faster-executing original application. In general, the distribution indicates that the obfuscation does not impose a consistent memory penalty on the system.

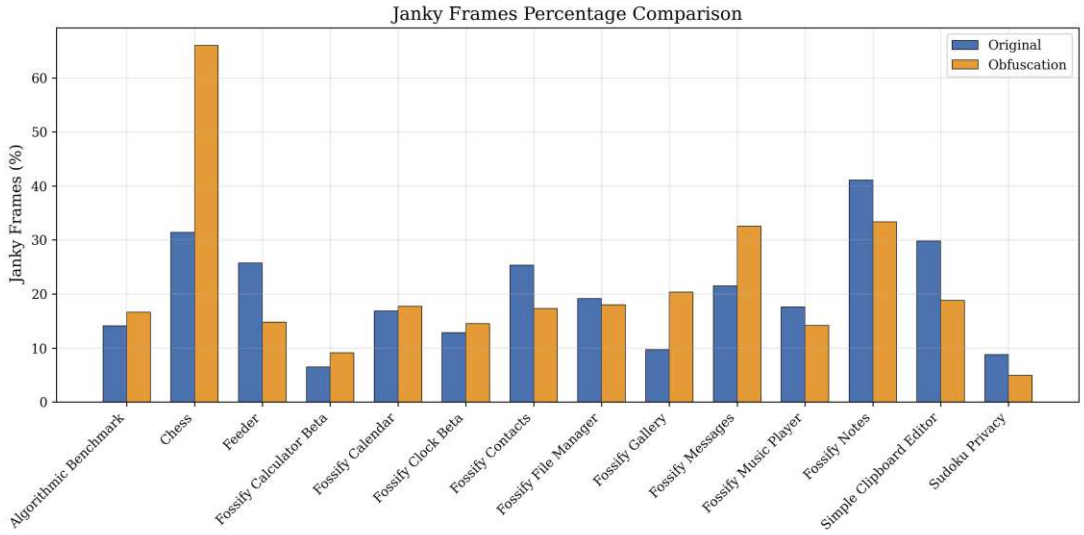

Figure 6.1: Comparison of janky frames between original and obfuscated applications.

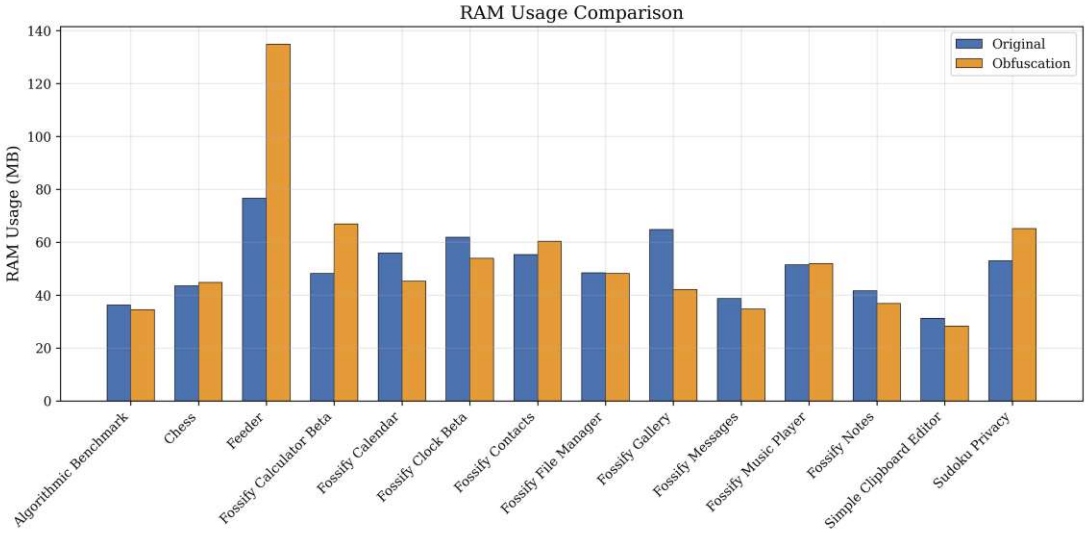

Figure 6.2: Comparison of RAM usage (MB) between original and obfuscated applications.

6.4.3 Execution Throughput

The experiments with the *Algorithmic Benchmark* reveals a consistent performance overhead across all experiments, ranging from 11.13% to 29.84%. Figure 6.3 illustrates these results.

The iterative Fibonacci implementation (n=10,000) shows the highest performance impact (29.84%), where the mean execution time increases from 1.066s to 1.385s. The recursive Fibonacci implementation (n=30) shows a 22.87% overhead, with execution time increasing from 9.848s to 12.100s. Finally, BubbleSort (n=1000) demonstrates the lowest overhead at 11.13% with a increase from 0.4008s to 0.4540s.

The performance degradation correlates directly with the mandatory opcode lookup inherent to the custom interpreter. For every instruction, the interpreter queries the permutation map to decode the instruction. This process introduces a constant time penalty to the fetch-decode-execute cycle. The overhead persists even for identity-mapped instructions, such as control flow opcodes, as the interpreter must still verify the mapping before it dispatches the opcode.

We observe that the relative overhead varies inversely with the computational complexity of the executed instructions. BubbleSort utilizes array-access instructions such as `AGET` and `APUT`. These operations are computationally heavier than simple arithmetic, which masks the fixed latency of the opcode lookup. In contrast, the Fibonacci benchmarks primarily execute lightweight instructions such as `ADD_INT` and `SUB_INT`.

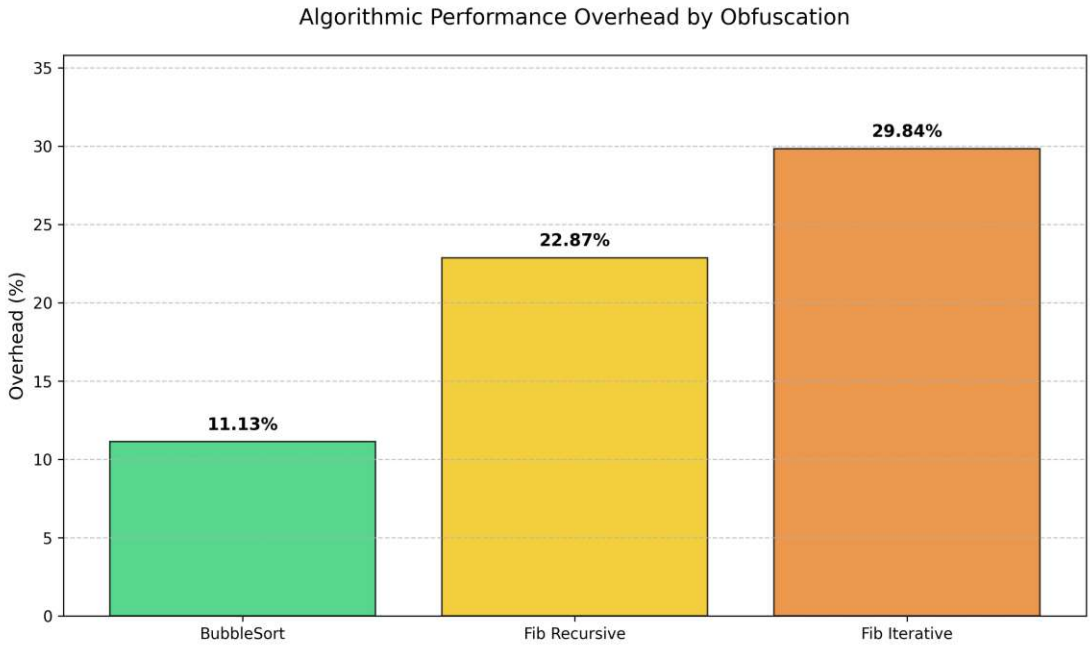

Figure 6.3: Performance overhead for each algorithmic benchmarks after obfuscation.

For these atomic operations, the lookup latency constitutes a significant fraction of the total execution time. Consequently, the iterative Fibonacci implementation exhibits the highest overhead. Its dense loop structure maximizes instruction throughput and exposes the decoding latency more frequently than the recursive version, which is bounded by the overhead of stack frame management and method invocations.

6.4.4 Launch Latency

Figure 6.4 presents the application launch time between original applications and the obfuscated counterparts. The data indicates that opcode permutation maintains a performance profile nearly identical to the unmodified environment for the majority of the test suite. Applications such as *Chess* and the *Algorithmic Benchmark* demonstrate negligible deltas, with variations below 20 ms.

A significant portion of the dataset, including *Fossify Messages* and *Fossify Gallery*, shows slightly improved launch times in the obfuscated state. These fluctuations do not suggest a performance gain from the obfuscation itself, but rather reflect the execution variance and caching behavior of the Android OS during warm-start cycles. Because the custom interpreter maps permuted opcodes to their original logic via a static lookup table, the instruction execution frequency remains constant.

Fossify Calendar presents a notable outlier in the performance dataset. The application has a performance degradation of approximately 80%, as the launch time increases from 2803 ms to 5045 ms. This specific latency correlates with the application's extensive resource binding and complex class loading sequences during the initial activity transition.

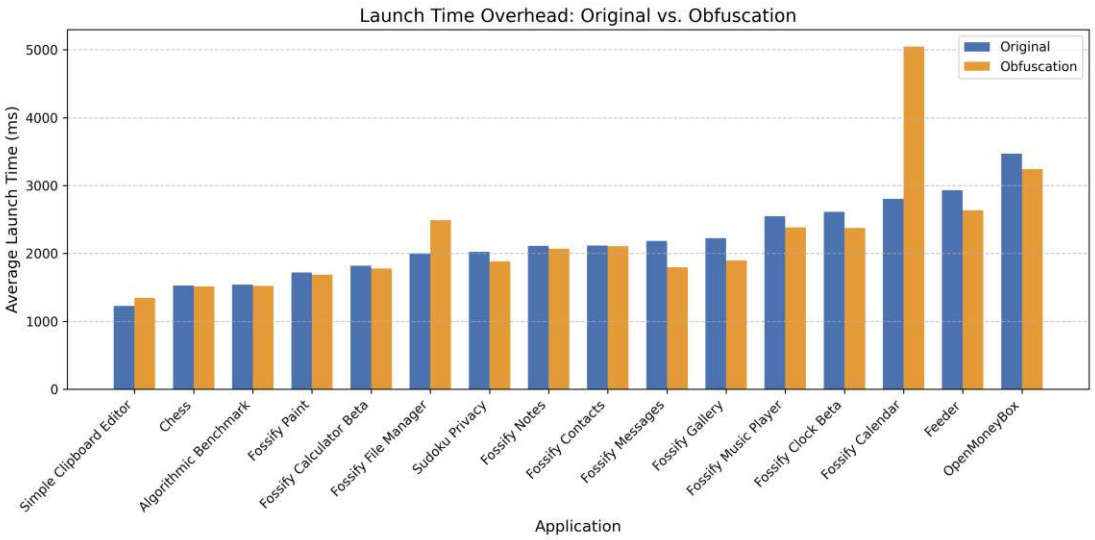

Figure 6.4: Comparison of average launch times (ms) for original and obfuscated applications.

These initialization patterns interact inefficiently with the modified interpreter's lookup mechanism and the processing of the manifest meta-flag. When we exclude this single outlier from the dataset, the average execution overhead across the remaining test suite becomes negligible.

6.4.5 Disk Space

Across the test set, the obfuscation introduces a negligible storage overhead. The median size increase for whole APKs is 2.17%, while the DEX files alone show a median growth of only 0.08%. Figure 6.5 illustrates the relative APK file size changes after obfuscation, while Figure 6.6 shows the relative DEX file size changes after obfuscation. The majority of applications have only minor increases, but some outliers exist within the dataset.

First, the *FREE Browser* application, which displays a significant reduction in APK size (-19.74%) and a DEX size increase of 0.08%. Second, our *Algorithmic Benchmark* app exhibits a reduction in APK size (-17.75%) and a DEX size decrease of -3.93%. The significant reduction in overall APK size is largely a side effect of using an uncompressed debug build. Similarly, the slight decrease in DEX size may occur because the disassembly and reassembly process streamlines the file structure and strips debug metadata. This supports the claim that our obfuscation technique has minimal growth impact and the original application packages likely utilized lower compression ratios or stored resources uncompressed.

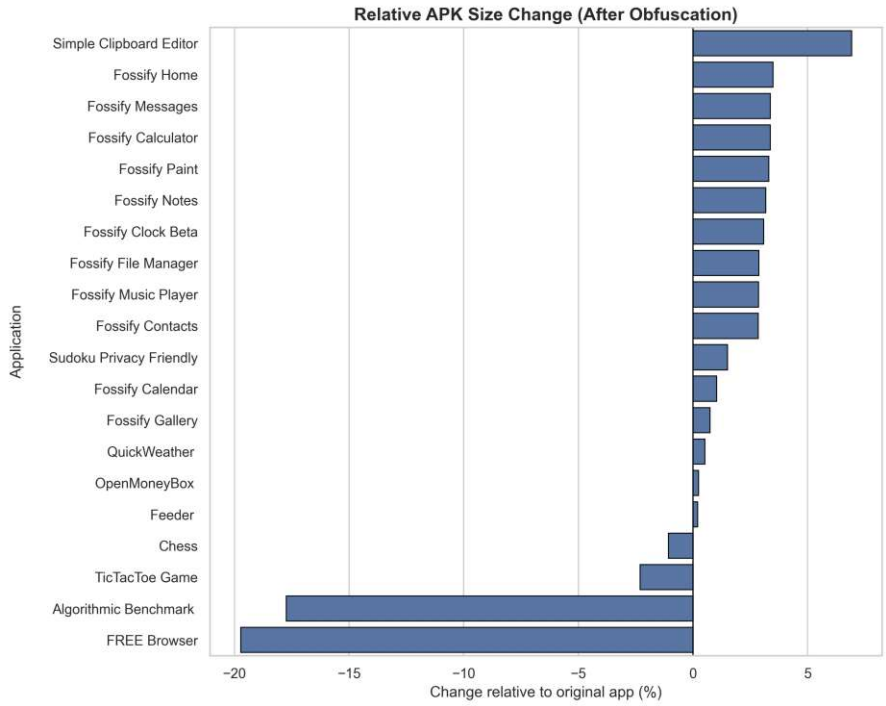

Figure 6.5: APK file size changes relative to original apps

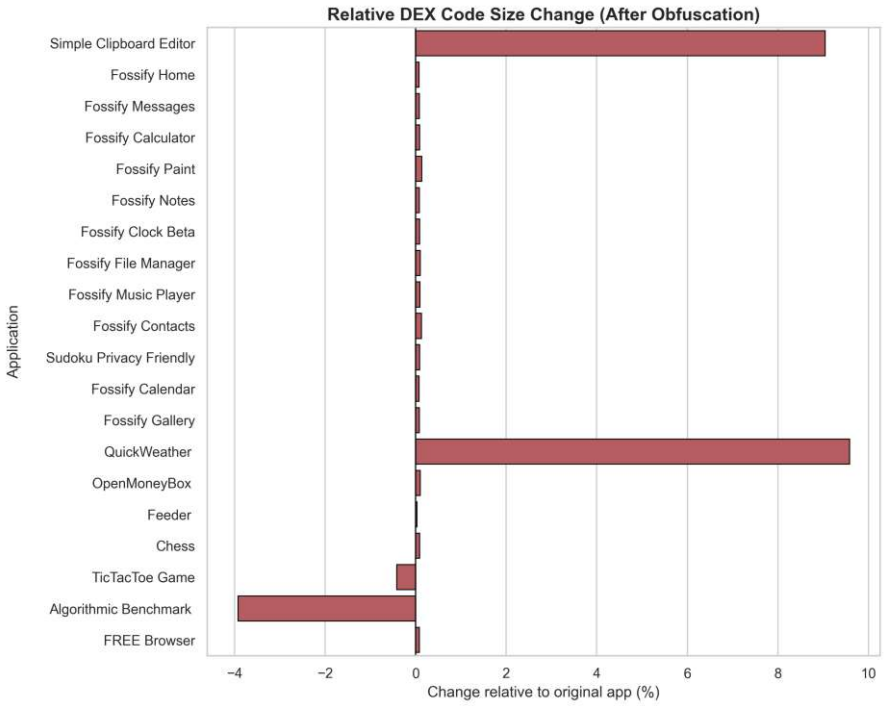

Figure 6.6: DEX file size changes relative to original apps

Simple Clipboard Editor and *QuickWeather* show a DEX size increases of 9.04% and 9.58%. This growth may be an artifact of the re-serialization process within the obfuscation pipeline, since the `dexlib2` library rewrites the dex files without optimizations. For the *Simple Clipboard Editor* (only ~16KB), the relative percentage is skewed by the re-introduction of standard headers or alignment padding.

There are three primary factors for the growth of file size. First, we always have a small overhead because of the injection of our own boolean flag `art.permutation.obfuscation` into the `AndroidManifest.xml`. Second, the permutation of opcodes modifies the entropy profile of the bytecode. This variation influences the compression efficiency of the DEFLATE algorithm within the ZIP container, which results in fluctuations in the final archive size. Finally, the repackaging pipeline applies Android v1, v2, and v4 signatures to ensure broad compatibility. While the v4 signature resides in a detached file, the v1 JAR signature artifacts and the v2 signing block append data directly to the APK. These cryptographic structures form the dominant fixed overhead, a factor that is particularly visible in smaller applications.

The complete experiment details can be seen in Table A.6 (disk space before obfuscation), Table A.7 (disk space after obfuscation) and Table A.8 (disk space percentage change).

6.4.6 Decompilation Failure Rate

Figure 6.7 illustrates the resilience of the obfuscated applications against static analysis. We classify the output into two categories: *Deceptive Code (green)* and *Broken Code (red)*. Deceptive Code represents methods that JADX reconstructs into syntactically valid Java source despite the underlying opcode permutation. Broken Code denotes methods that trigger fatal decompilation errors.

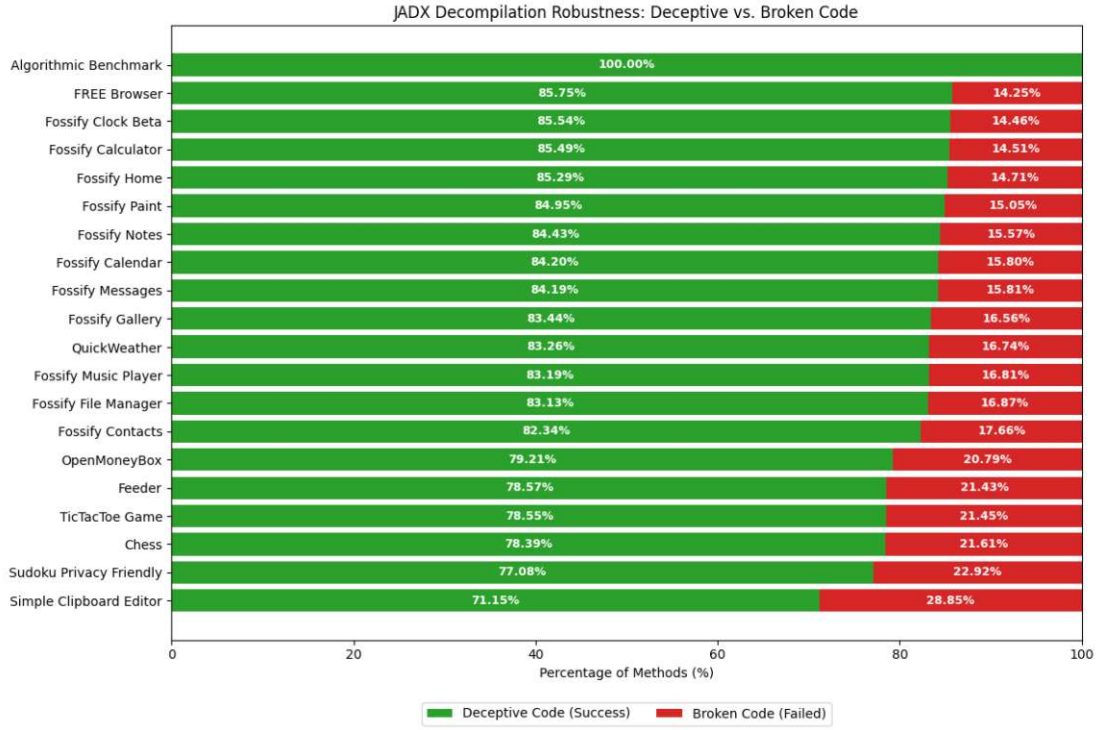

Figure 6.7: Source reconstruction rate of methods from JADX decompilation.

The data reveals a high success rate across all applications, which ranges from 71.15% (*Simple Clipboard Editor*) to 100% (*Algorithmic Benchmark*). These results indicate that the majority of opcode swaps do not disrupt the structural integrity required for AST generation. Consequently, the permutation technique effectively creates a "stealth" layer of misleading source code rather than forcing a complete tool failure. However, extraction failures persist with an average rate of approximately 20%. This demonstrates that specific opcode permutations successfully destabilize the control flow analysis of the decompiler. The *Feeder* application exhibits the highest error frequency, peaking at 34,808 total errors. On average, the obfuscation induces an error density of 2.5 errors per failed method.

Figure 6.8 shows the volume of internal JADX errors. Obfuscated applications trigger a substantial increase in system log entries compared to the baseline. Original applications produce negligible error counts, with log counts ranging from 0 to 31 and a mean of

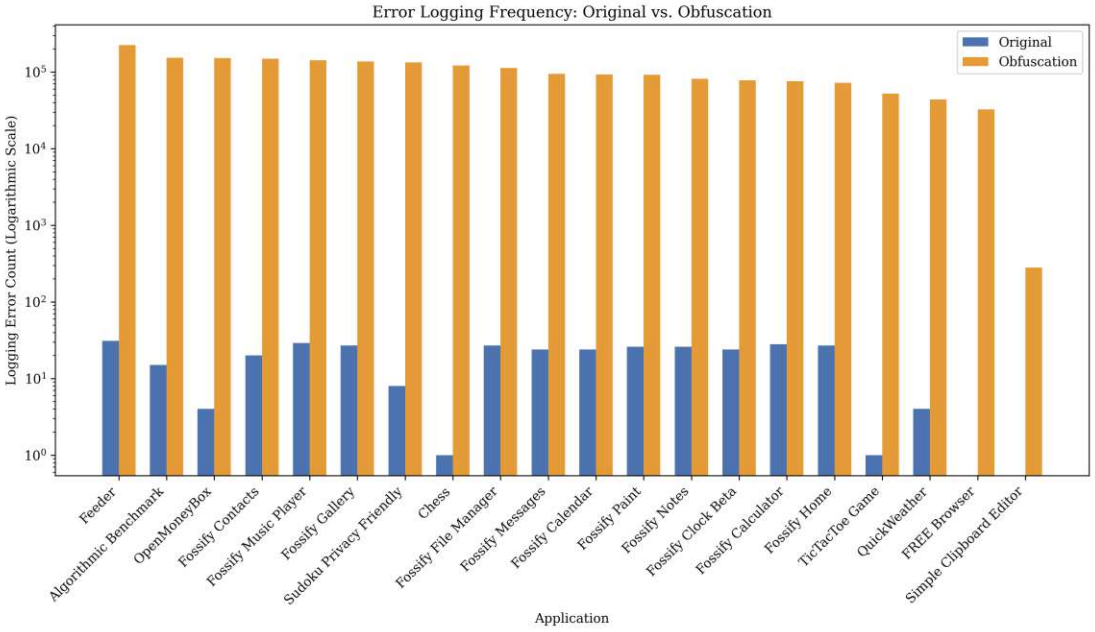

Figure 6.8: Comparison of JADX error logging frequency between original and obfuscated applications.

17.3 errors. In contrast, obfuscated versions generate between 279 and 223,644 entries, with a mean of 102,082.7 entries. This surge in log activity quantifies the struggle of the decompilation engine to process the permutated bytecode.

For a complete breakdown of decompilation metrics for all 20 tested applications, refer to Table A.1.

6.4.7 Readability & Compatibility with Traditional Obfuscations

Listing 20 shows the decompiled '`bubbleSort`' method from the original *Algorithmic Benchmark* application. The method was successfully decompiled and is clearly readable. This pretty view shows the high fidelity decompilation of JADX, when dealing with non obfuscated applications.

Figure 6.9 shows the comparison and impact of our obfuscation on application. Figure 6.9a presents the original `bubbleSort` method in the standard register-based fallback view. The decompiler successfully reconstructs the logic and produces readable instructions. Conversely, Listing 6.9b displays the result after opcode permutation. JADX generates syntactically valid Java code, yet the underlying semantics differ completely from the original logic. Line 3 exemplifies this corruption: the original `int r0 = r8.length` instruction appears as `long r0 = r0 << r8` in the obfuscated version. This semantic distortion prevents standard static analysis tools from deriving the correct program behavior without the corresponding permutation mapping.

```java
public final void bubbleSort(int[] arr) {
    int n = arr.length;
    int i = n - 1;
    for (int i2 = 0; i2 < i; i2++) {
        int i3 = (n - i2) - 1;
        for (int j = 0; j < i3; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

Listing 20: Original bubbleSort method (pretty view) from JADX decompilation.

```java
public final void bubbleSort(int[] r8) {
    r7 = this;
    int r0 = r8.length
    r1 = 0
    int r2 = r0 + (-1)
L4:
    if (r1 >= r2) goto L27
    r3 = 0
    int r4 = r0 - r1
    int r4 = r4 + (-1)
Lb:
    if (r3 >= r4) goto L24
    r5 = r8[r3]
    int r6 = r3 + 1
    r6 = r8[r6]
    if (r5 <= r6) goto L21
    r8[r3] = r6
    int r6 = r3 + 1
    r8[r6] = r5
L21:
    r3 = r3 + 1
    goto Lb
L24:
    r1 = r1 + 1
    goto L4
L27:
    return
}
```

(a) Original method (fallback view)

```java
public final void bubbleSort(int[] r8) {
    r7 = this;
    long r0 = r0 << r8
    r1 = 0
    r2 = r0 ^ (-1)
L4:
    if (r1 >= r2) goto L27
    r3 = 0
    r0[r1] = r4
    r4 = r4 ^ (-1)
Lb:
    if (r3 >= r4) goto L24
    r8[r3] = r5
    r6 = r3 ^ 1
    r8[r6] = r6
    if (r5 <= r6) goto L21
    r8[r3] = r5
    r6 = r3 ^ 1
    r5 = r8 + r6
L21:
    r3 = r3 ^ 1
    goto Lb
L24:
    r1 = r1 ^ 1
    goto L4
L27:
    goto Lf
}
```

(b) Method with opcode permutation

Figure 6.9: Impact of opcode permutation on JADX decompilation. (a) shows the standard register-based view for the original app, while (b) demonstrates how opcode permutation corrupts the semantic interpretation.

Listing 21 demonstrates the benefits of using multiple obfuscation strategies at the same time. In addition to the opcode permutation, we use traditional obfuscation techniques such as identifier renaming to obscure fields and method names, as well as junk code insertion. The traditional obfuscations have to be applied before the opcode permutation, or else we would break the permutation obfuscation. The additional benefit is that the junk code instructions are also swapped by our obfuscation as a last step. As a result, we see that the 'bubbleSort' method name is now random gibberish and the first lines have additional code. This compounding effect significantly elevates the barrier to entry for reverse engineering compared to using any single technique in isolation.

```
1  private final void mb9fc1d9c(int[] r8) {  // Identifier Renaming
2      r7 = this;
3      r0 = 19;
4      r1 = 12;
5      int r0 = r0 / r1;    // Junk Code Insertion (useless arithmetic operations)
6      r0 = r0[r1];         // Invalid: Array access on int
7      if (r0 > 0) goto Lf;
8      goto L37;
9  Lf:
10     long r0 = r0 % r8;  // Invalid: Modulo on array object
11     r1 = 0;
12     int r2 = r0 << (-1);
13 L13:
14    // ... (remaining code) ...
15 }
```

Listing 21: Decompiled output showcases the cumulative effect of identifier renaming, junk code insertion, and opcode permutation.

6.4.8 Control Flow Graph Distortion

Ghidra generates error logs during analysis similar to JADX but still allows code exploration. We successfully locate the `runBenchmarks` method within the obfuscated *Algorithmic Benchmark* application. This identification remains feasible because the current obfuscation scope is strictly limited to bytecode instructions and excludes identifier renaming or string encryption. Figure 6.10 presents the generated CFGs of the method.

The original method (Figure 6.10a) consists of 7 vertices and 9 edges. In contrast, the graph generated from the obfuscated bytecode (Figure 6.10b) shows distinct structural divergence, expanding to 10 vertices and 13 edges. This modification occurs even though the obfuscation logic explicitly excludes control flow instructions.

The observed structural corruption stems from disassembly desynchronization. The opcode swapping mechanism introduces instruction sequences that violate the standard Dalvik bytecode format. These anomalies cause the analysis tool to lose synchronization with legitimate instruction boundaries, which leads to byte-alignment errors.

(a) Method before obfuscation

(b) Method after obfuscation

Figure 6.10: Comparison of Ghidra Function Graphs for the `runBenchmarks` method in the *Algorithmic Benchmark* application. (a) shows the valid control flow of the original app. (b) shows the distorted graph after opcode permutation, where disassembly errors generate artificial basic blocks and edges.

Consequently, Ghidra misinterprets non-control opcodes or immediate data as branching instructions. Furthermore, the parser's inability to resolve invalid opcodes forces the generation of disconnected error blocks, which artificially inflates the vertex count.

6.4.9 Dynamic Instrumentation Resistance

The opcode permutation mechanism prevents execution on standard Android Runtime environments. Attempts to launch the obfuscated applications on stock Android emulators or physical devices result in immediate process termination. The standard interpreter fails to decode the permuted bytecode and triggers fatal errors or verification failures (e.g., `VerifyError`) before the application entry point is reached.

This incompatibility serves as a robust defense against automated sandboxing and off-the-shelf dynamic analysis frameworks. Tools that rely on standard Android environments to hook or trace execution flow are rendered ineffective, as the application logic remains

inaccessible outside of the specific execution context. Consequently, any successful dynamic instrumentation or runtime analysis requires the custom ART capable of interpreting the permuted instruction set.

6.5 Discussion

The experimental results validate that the proposed opcode permutation technique successfully shifts the obfuscation paradigm from metadata manipulation to instruction set virtualization. Addressing the compatibility with traditional techniques (**RQ1**), our analysis of layered obfuscation confirms that opcode permutation operates orthogonally to standard methods. When integrated with identifier renaming and junk code insertion, the technique achieves a compounding defense effect. The permutation of inserted junk code specifically prevents analysts from easily distinguishing between valid logic and obfuscated noise, thereby significantly elevating the reverse engineering barrier beyond what either technique achieves in isolation.

Regarding the implementation requirements for the Android Runtime (**RQ2**), the compatibility analysis highlights that a successful execution environment relies on more than just opcode mapping. The crash in the *TicTacToe Game* indicates that the runtime modification must seamlessly support hybrid contexts where bytecode logic is tightly coupled with native asset loading. In addition, the failures in *Free Browser* and *Quick-Weather* demonstrate that the custom ART must ensure strict structural consistency. For example, the runtime requires that swapped opcodes not only adhere to compatible format groups and layout constraints, but also share register widths. Furthermore, the results underscore the complexity of ensuring complete handler coverage within the Android Runtime, where failing to intercept specific instruction sequences or edge cases leads to immediate execution failures.

Finally, the evaluation confirms the technique's effectiveness in fortifying applications (**RQ3**) while balancing quality criteria. The obfuscation successfully disrupts static analysis, as evidenced by JADX generating deceptive, semantically incorrect code and Ghidra suffering from disassembly desynchronization that distorts the control flow graph. The inherent incompatibility with stock environments provides a robust defense against dynamic code instrumentation. These security gains show a performance trade-off: while storage overhead is negligible (0.08% median DEX growth) and launch times remain stable, the mandatory opcode lookup introduces a consistent execution throughput penalty of up to 29.84% in instruction-dense scenarios. This defines the technique as a highly effective protection mechanism for security-critical logic where storage efficiency is prioritized over raw computational speed.

CHAPTER 7

Conclusion and Future Work

In this thesis, we presented a system-level virtualization framework to protect Android applications against reverse engineering. Unlike application-level virtualization that embeds custom interpreters and suffers from JNI overhead, this approach modifies the Android Runtime to support Instruction Set Randomization directly. The solution consists of the Android Opcode Permutation Tool (AOPT) and a modified Android operating system. We structure our findings by addressing the three defined research questions.

RQ1 - *What are the most common traditional code obfuscation techniques, and how can these insights be leveraged to design advanced obfuscation techniques within the Android ecosystem?*

We identify identifier renaming, control flow flattening, and encryption as the most prevalent traditional obfuscation techniques. In addition, runtime protections utilize dynamic code loading and reflection to hide the payload until execution, while mechanisms such as anti-debugging actively prevents dynamic analysis. We observe that, despite their functional differences, all these techniques operate exclusively on the application level and rely on a standard execution environment. We leverage this insight by shifting the obfuscation scope from the application code to the Android Runtime. Consequently, this system-level approach invalidates the semantic assumptions of reverse engineering tools regarding the instruction set. Furthermore, we confirm that traditional application-level obfuscation techniques remain orthogonal to our virtualization-based obfuscation and achieve a robust, layered protection effect when combined.

RQ2 - *How can the Android Runtime be modified to support a state-of-the-art virtualization-based obfuscation technique, and what requirements are necessary for a successful implementation?*

We demonstrate that extending the Android Runtime is feasible and effective. A successful implementation requires a metadata flag to signal the obfuscation status and

the integration of a secondary interpreter to execute the randomized bytecode. AOPT permutes Dalvik bytecode into a unique, randomized instruction set for each application and injects a metadata flag into the app manifest. The modified runtime detects this flag and activates a custom `C++` switch interpreter. The switch interpreter performs dynamic opcode lookups for assertion checks and utilizes macro-generated switch cases to execute the randomized instruction set. This design preserves compatibility with standard applications while enforcing the obfuscation logic only for secured apps.

RQ3 - *How effective is the implemented virtualization-based obfuscation technique in fortifying the security of Android applications against various methods of reverse engineering, and what quality criteria are affected by applying the obfuscation?*

Our evaluation demonstrates that the obfuscation proves highly effective against standard reverse engineering tools. Static analysis with JADX generates broken and deceptive code. Some methods decompile into valid Java syntax but contain semantic errors, such as interpreting array length as a bitwise shift instruction. Ghidra suffers from disassembly desynchronization, which results in distorted control flow graphs and incorrect edge counts. Dynamic analysis tools such as Frida fail immediately because they cannot interpret the randomized instruction set. However, it is essential to acknowledge that no obfuscation scheme is unbreakable. The system-level virtualization presented here significantly raises the bar for attackers by invalidating standard tools, but it acts as a deterrent rather than a guarantee. Given sufficient resources and time, a determined adversary can eventually reverse engineer the obfuscated code.

Our performance evaluation reveals a consistent overhead for algorithmic tasks ranging from 11.13% to 29.84%. This results from the mandatory opcode lookup during the fetch-decode-execute cycle. However, launch latency remains stable for most apps with variations below 20ms, and storage overhead proves negligible with a median APK size increase of only 2.17%. The modified execution environment achieves an 85% compatibility rate for apps. Failures primarily arise from edge cases involving JNI boundaries or register width mismatches.

Beyond these functional limitations, the proposed system-level virtualization has inherent deployment constraints. Obfuscated applications cannot be executed on standard devices and are coupled to the implementation of specific AOSP versions. This dependency blocks distribution through official channels such as the Google Play Store.

Therefore, this system-level virtualization obfuscation is best suited for high-security environments where the organization retains strict control over the operating system firmware. Rather than targeting mass-market consumer applications, this system is designed for government agencies and corporations protecting high-value targets. In these scenarios, the capability to secure devices against advanced reverse engineering outweighs the loss of portability and justifies the deployment of custom Android images.

7.1 Future Work

Future research should prioritize mitigating execution overhead by integrating the opcode permutation logic into high-performance ART components. Modern Android utilizes the assembly-based `nterp` interpreter to minimize fetch-decode-execute latency. We propose extending `nterp` to support randomized instruction sets, which would eliminate the performance penalty associated with the `C++` switch interpreter. To further close the performance gap, future work should also explore modifications to the JIT and `dex2oat` compilers. These adaptations allow obfuscated apps to execute as optimized native code and restore the performance profile of standard applications. Future research should also explore dynamic instruction set mutations to rotate mappings at runtime.

Beyond performance enhancements, a sustainable maintenance strategy is required to address the rapid release cycle of the Android Open Source Project. The current implementation involves deep modifications to the interpreter and verifier, which create strong dependencies on specific Android versions. Future work must establish an automated patching system or a modular abstraction layer that separates the obfuscation logic from the underlying runtime implementation. This would reduce the engineering effort required to port the virtualization environment to future Android releases and ensures long-term viability.

APPENDIX A

Detailed Experimental Results

A.1 Decompilation Failure Rate

Application	Total Methods	Failed Methods	Success Rate	Total Errors	Logging Errors
Algorithmic Benchmark	49,506	0	100%	0	152,867
Chess	20,405	4,142	79.70%	8,970	121,375
Feeder	63,714	11,384	78.57%	34,808	223,644
Fossify Calendar	28,154	4,220	85.01%	9,143	92,850
Fossify Calculator	20,812	4,142	80.10%	8,961	75,348
Fossify Clock Beta	25,626	4,166	83.74%	9,030	77,487
Fossify Contacts	23,246	4,161	82.10%	8,997	149,488
Fossify File Manager	28,092	4,228	84.95%	9,165	113,282
Fossify Gallery	34,769	4,375	87.42%	10,757	137,031
Fossify Home	17,992	4,142	76.98%	8,981	72,475
Fossify Messages	22,230	4,153	81.32%	8,984	94,791
Fossify Music Player	30,359	4,249	86.00%	9,219	142,686
Fossify Notes	25,504	4,154	83.71%	8,980	81,671
Fossify Paint	20,832	4,142	80.12%	8,962	92,101
FREE Browser	40,028	6,838	82.92%	17,738	32,410
OpenMoneyBox	65,973	10,794	79.21%	24,789	152,147
QuickWeather	16,210	2,167	83.26%	5,801	43,852
Simple Clipboard Editor	52	15	71.15%	49	279
Sudoku Privacy Friendly	42,912	9,097	78.80%	25,233	133,855
TicTacToe Game	21,846	4,135	78.55%	8,951	52,015

Table A.1: JADX decompilation error metrics for obfuscated applications.

A.2 Stress Testing

Application	Status	RAM (kB)	Total Frames	Janky Frames	Janky %
Algorithmic Benchmark	Success	37,138	950	134	14.11%
Chess	Success	44,619	1,957	614	31.37%
Feeder	Success	78,511	101	26	25.74%
Fossify Calculator Beta	Success	49,407	2,437	158	6.48%
Fossify Calendar	Success	57,283	2,662	448	16.83%
Fossify Clock Beta	Success	63,348	5,871	753	12.83%
Fossify Contacts	Success	56,711	2,359	597	25.31%
Fossify File Manager	Success	49,654	1,711	328	19.17%
Fossify Gallery	Success	66,370	2,897	280	9.67%
Fossify Messages	Success	39,678	1,754	377	21.49%
Fossify Music Player	Success	52,755	2,085	367	17.60%
Fossify Notes	Success	42,702	2,016	828	41.07%
Simple Clipboard Editor	Success	32,041	2,172	647	29.79%
Sudoku Privacy	Success	54,289	3,847	338	8.79%

Table A.2: Monkey stress testing results (5,000 events) for original applications.

Application	Status	RAM (kB)	Total Frames	Janky Frames	Janky %
Algorithmic Benchmark	Success	35,315	1,039	173	16.65%
Chess	Success	45,915	2,510	1,656	65.98%
Feeder	Success	138,086	2,577	381	14.78%
Fossify Calculator Beta	Success	68,506	6,796	621	9.14%
Fossify Calendar	Success	46,397	3,407	603	17.70%
Fossify Clock Beta	Success	55,231	5,432	789	14.53%
Fossify Contacts	Success	61,880	3,028	524	17.31%
Fossify File Manager	Success	49,418	1,786	321	17.97%
Fossify Gallery	Success	43,155	1,604	326	20.32%
Fossify Messages	Success	35,642	1,445	470	32.53%
Fossify Music Player	Success	53,219	2,235	317	14.18%
Fossify Notes	Success	37,741	1,169	390	33.36%
Fossify Paint	Crash	-	-	-	-
FREE Browser	Crash	-	-	-	-
OpenMoneyBox	Crash	-	-	-	-
Simple Clipboard Editor	Success	28,985	2,254	424	18.81%
Sudoku Privacy	Success	66,710	4,328	213	4.92%

Table A.3: Monkey stress testing results (5,000 events) for obfuscated applications.

A.3 Launch Latency

Application	Launch Measurements (ms)					
	T1	T2	T3	T4	T5	Mean
Algorithmic Benchmark	1538	1532	1535	1603	1497	1541
Chess	1589	1545	1484	1522	1495	1527
Feeder	3180	2500	3297	2561	3116	2930
Fossify Calculator Beta	1751	1771	1770	1842	1974	1821
Fossify Calendar	3080	2284	2899	2849	2907	2803
Fossify Clock Beta	2700	2578	2607	2688	2495	2613
Fossify Contacts	2370	2208	1917	2023	2059	2115
Fossify File Manager	2078	1973	2077	1897	1969	1998
Fossify Gallery	2127	2253	2335	2189	2226	2226
Fossify Messages	2523	2080	2083	2164	2071	2184
Fossify Music Player	2471	2563	2554	2636	2527	2550
Fossify Notes	2254	1929	2071	2250	2062	2113
Fossify Paint	1715	1654	1767	1761	1711	1721
OpenMoneyBox	3414	3297	3635	3519	3493	3471
Simple Clipboard Editor	1255	1146	1274	1353	1115	1228
Sudoku Privacy	2135	1651	2281	1819	2244	2026

Table A.4: Application cold start launch times (ms) across five iterations.

Obfuscated Application	Launch Measurements (ms)					
	T1	T2	T3	T4	T5	Mean
Algorithmic Benchmark	1483	1644	1453	1562	1476	1523
Chess	1514	1615	1487	1450	1500	1513
Feeder	2897	2743	2467	2327	2750	2636
Fossify Calculator Beta	1742	1757	1820	1782	1790	1778
Fossify Calendar	5255	4770	4990	5144	5068	5045
Fossify Clock Beta	2419	2314	2465	2352	2324	2374
Fossify Contacts	2088	2071	2222	2062	2091	2106
Fossify File Manager	2377	2551	2535	2534	2459	2491
Fossify Gallery	1879	1853	1957	1940	1867	1899
Fossify Messages	1776	1756	1897	1822	1737	1797
Fossify Music Player	2342	2362	2497	2343	2373	2383
Fossify Notes	2044	2069	2091	2057	2086	2069
Fossify Paint	1600	1656	1783	1737	1660	1687
OpenMoneyBox	3621	3547	3033	3073	2929	3240
Simple Clipboard Editor	1317	1355	1345	1373	1354	1348
Sudoku Privacy	1662	2090	2231	1642	1789	1882

Table A.5: Obfuscated application cold start launch times (ms) across five iterations.

A.4 Disk Space

Application	APK Size (KB)	Total DEX Size (KB)	DEX File Count
TicTacToe Game	4370.06	3359.26	1
OpenMoneyBox	30210.73	11557.53	3
Feeder	60670.57	11767.21	2
QuickWeather	17374.17	2997.34	1
Simple Clipboard Editor	80.46	16.92	1
Chess	7036.24	9217.89	2
Algorithmic Benchmark	14033,46	10379,15	3
Fossify Calendar	8479.42	5082.81	1
Fossify Clock Beta	8823.33	4683.00	1
Fossify Contacts	8331.78	7575.85	2
Fossify File Manager	9689.45	6523.62	1
Fossify Gallery	37657.70	7647.29	1
Fossify Home	6432.79	4428.16	1
Fossify Calculator	6738.95	4482.91	1
Fossify Messages	8010.99	5166.93	1
Fossify Music Player	10678.79	7067.84	1
Fossify Notes	9227.98	4572.40	1
Fossify Paint	7356.32	5859.11	1
Sudoku Privacy Friendly	5071.13	9829.16	2
FREE Browser	5289.66	2330.50	1

Table A.6: APK and DEX sizes (in KB) before obfuscation.

Application	**APK Size (KB)**	**DEX Size (KB)**	**DEX File Count**
TicTacToe Game	4268.94	3345.17	1
OpenMoneyBox	30282.96	11568.75	3
Feeder	60788.67	11770.08	2
QuickWeather	17463.38	3284.51	1
Simple Clipboard Editor	86.02	18.45	1
Chess	6960.65	9225.87	2
Algorithmic Benchmark	11542,72	9971,08	3
Fossify Calendar	8565.69	5086.57	1
Fossify Clock Beta	9094.95	4687.06	1
Fossify Contacts	8568.52	7585.38	2
Fossify File Manager	9967.57	6530.01	1
Fossify Gallery	37934.17	7653.43	1
Fossify Home	6656.79	4431.32	1
Fossify Calculator	6965.48	4486.57	1
Fossify Messages	8280.64	5170.84	1
Fossify Music Player	10983.87	7074.46	1
Fossify Notes	9519.63	4575.79	1
Fossify Paint	7598.74	5866.71	1
Sudoku Privacy Friendly	5146.75	9837.19	2
FREE Browser	4245.52	2332.32	1

Table A.7: APK and DEX sizes (in KB) after obfuscation.

Application	APK Size Δ%	DEX Size Δ%
Simple Clipboard Editor	+6.91%	+9.04%
Fossify Home	+3.48%	+0.07%
Fossify Messages	+3.37%	+0.08%
Fossify Calculator	+3.36%	+0.08%
Fossify Paint	+3.30%	+0.13%
Fossify Notes	+3.16%	+0.07%
Fossify Clock Beta	+3.08%	+0.09%
Fossify File Manager	+2.87%	+0.10%
Fossify Music Player	+2.86%	+0.09%
Fossify Contacts	+2.84%	+0.13%
Sudoku Privacy Friendly	+1.49%	+0.08%
Fossify Calendar	+1.02%	+0.07%
Fossify Gallery	+0.73%	+0.08%
QuickWeather	+0.51%	+9.58%
OpenMoneyBox	+0.24%	+0.10%
Feeder	+0.19%	+0.02%
Chess	-1.07%	+0.09%
TicTacToe Game	-2.31%	-0.42%
Algorithmic Benchmark	-17.75%	-3.93%
FREE Browser	-19.74%	+0.08%

Table A.8: Disk space analysis shows the relative percentage change (Δ%) after obfuscation.

APPENDIX B

Additional Material

B.1 Dalvik Executable Format Reference

Name	Type	Description
header	header_item	Metadata (checksum, version, size) and offsets mapping the file structure.
string_ids	string_id_item[]	Identifiers for all strings used for naming and constants.
type_ids	type_id_item[]	Identifiers for all referenced types (classes, arrays, primitives).
proto_ids	proto_id_item[]	Method prototypes defining return types and parameters.
field_ids	field_id_item[]	Field identifiers describing the defining class, type, and name.
method_ids	method_id_item[]	Method identifiers linking class, name, and prototype.
class_defs	class_def_item[]	Class definitions (access flags, superclasses, interfaces, data pointers).
call_site_ids	call_site_id_item[]	Identifiers for call sites (dynamic invocation).
method_handles	method_handle_item[]	List of referenced method handles.
data	ubyte[]	Raw data pool containing bytecode, static values, and annotations.
link_data	ubyte[]	Reserved space for static linking data (typically unused).

Table B.1: Dalvik executable format layout [89]

B.2 AOSP Build and Runtime Configuration

This section describes the commands to download, compile, and configure the Android Open Source Project, as well as the emulator configuration.

Repository Setup

We utilize the `repo` tool to manage the Git repositories of the AOSP.

```
mkdir -p ~/.bin
PATH="${HOME}/.bin:${PATH}"
curl https://storage.googleapis.com/git-repo-downloads/repo > ~/.bin/repo
chmod a+rx ~/.bin/repo
```

We initialize the repository using the specific tag `android-15.0.0_r14` and synchronize the source tree.

```
repo init -u https://android.googlesource.com/platform/manifest
          -b android-15.0.0_r14

repo sync -c -j8
```

Build Configuration

We configure the build target for an `x86_64` emulator engineering build.

```
# Initialize the build environment
source build/envsetup.sh

# Select the target architecture
lunch sdk_phone64_x86_64-trunk_staging-eng

# Build the system
make
```

Emulator Execution

We execute the custom Android system using the standard emulator. We utilize the `-gpu host` flag to enable hardware acceleration and `-wipe-data` to ensure a clean state for each test run.

```
emulator -wipe-data -verbose -gpu host
```

Runtime Environment Configuration

To validate the opcode permutation, we must bypass the JIT/AOT compiler and the optimized Nterp interpreter. The following commands configure the runtime properties to force the execution path through the C++ switch interpreter.

```
# Gain root access
adb root

# Stop the runtime environment
adb shell stop

# Disable JIT compilation
adb shell setprop dalvik.vm.usejit false

# Set execution mode to the portable interpreter
adb shell setprop dalvik.vm.execution-mode int:fast

# Disable the Nterp interpreter
adb shell setprop dalvik.vm.enable_nterp false

# Restart the runtime environment
adb shell start
```

B.3 Hardware Specifications

Component	Specification
Processor (CPU)	AMD Ryzen 9 5950X (16-Core, 3.4 GHz)
Memory (RAM)	64 GB DDR4 3600 MHz
Storage	1TB SSD (Samsung 870 QVO)
Graphic Card (GPU)	NVIDIA GeForce RTX 3070 TI 8 GB

Table B.2: Hardware specifications of the host workstation.

Overview of Generative AI Tools Used

We utilized the following IDEs with their integrated AI-driven code completion features to support our software development process: IntelliJ IDEA[1], PyCharm[2], Android Studio[3] and Android Studio Platform[4].

Google Gemini[5] served as a supportive tool for data analysis and acted as a knowledge base for technical questions regarding the Android Open Source Project.

The thesis writing process was supported by both Google Gemini and DeepL Write[6] to refine the overall writing style. No generative AI tools were used to generate research findings or draw conclusions. All statistical analyses, interpretations, and arguments represent the author's own work.

[1] https://www.jetbrains.com/idea/
[2] https://www.jetbrains.com/pycharm/
[3] https://developer.android.com/studio
[4] https://developer.android.com/studio/platform
[5] https://gemini.google.com
[6] https://www.deepl.com

List of Figures

List of Code

Acronyms

ADB Android Debug Bridge. 52

AES Advanced Encryption Standard. 38

AID Android ID. 14

AOPT Android Opcode Permutation Tool. 51, 56, 61

AOSP Android Open Source Project. 5, 53

AOT Ahead-Of-Time. 9, 52

ART Android Runtime. 9, 26

AST Abstract Syntax Tree. 69

CFG Control Flow Graph. 24

DAC Discretionary Access Control. 13

DEX Dalvik Executable Format. 11

DVM Dalvik Virtual Machine. 8, 25

GDB GNU Debugger. 20

GID Group Identifier. 13

HAL Hardware Abstraction Layer. 7, 53

ISA Instruction Set Architecture. 48, 51, 55, 58

ISR Instruction Set Randomization. 51, 55

JDWP Java Debug Wire Protocol. 20, 41

JIT Just-In-Time. 8, 52

JNI Java Native Interface. 12, 26, 42, 60

JRE Java Runtime Environment. 54

JVM Java Virtual Machine. 8

LLDB Low Level Debugger. 20

LLMs Large Language Models. 31

NDK Native Development Kit. 12, 39

OAT Of-Ahead-Time. 9

TEE Trusted Execution Environment. 37

UID User Identifier. 13

Bibliography

[1] J. Xun, W. K. Chong, and L. Dolega. Mobile operating systems' impact on customer value: Ios vs. android. *Journal of Computer Information Systems*, 0(0):1–14, 2023.

[2] G. Meike and L. Schiefer. *Inside the Android OS - Building, Customizing, Managing and Operating Android System Services*. Addison Wesley, 2021. ISBN: 978-0-13-409634-6.

[3] N. Elenkov. *Android Security Internals - An In-depth Guide to Android's Security Architecture*. No Starch Press, 2014. ISBN: 978-1593275815.

[4] D. Chell, T. Erasmus, S. Colley, and O. Whitehouse. *The Mobile Application Hacker's Handbook*. Wiley, 2015. ISBN: 978-1-118-95850-6.

[5] R. Mayrhofer, J.V. Stoep, C. Brubaker, and N. Kralevich. The android platform security model. *ACM Trans. Priv. Secur.*, 24(3), 2021.

[6] J. Six. *Application Security for the Android Platform: Processes, Permissions, and Other Safeguards*. O'Reilly Media, 2011.

[7] H. Wang, H. Liu, X. Xiao, G. Meng, and Y. Guo. Characterizing android app signing issues. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 280–292, 2019.

[8] C. Negus. *Linux bible - Tenth Edition*. Wiley, 2020. ISBN: 978-1119578888.

[9] S. Smalley and R. Craig. Security enhanced (se) android: Bringing flexible mac to android. In *Proceedings of the Network and Distributed System Security Symposium*, volume 310, pages 20–38, 2013.

[10] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. Gaur, M. Conti, and M. Rajarajan. Android security: A survey of issues, malware penetration, and defenses. *IEEE Communications Surveys & Tutorials*, 17(2):998–1022, 2015.

[11] N. Mauthe, U. Kargén, and N. Shahmehri. A large-scale empirical study of android app decompilation. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 400–410, 2021.

[12] A. Merlo, A. Ruggia, L. Sciolla, and L. Verderame. You shall not repackage! demystifying anti-repackaging on android. *Computers and Security*, 103:102181, 2021.

[13] L. Glanz, S. Amann, M. Eichberg, M. Reif, B. Hermann, J. Lerch, and M. Mezini. Codematch: obfuscation won't conceal your repackaged app. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 638–648, New York, NY, USA, 2017. Association for Computing Machinery.

[14] T. Sutter, T. Kehrer, M. Rennhard, B. Tellenbach, and J. Klein. Dynamic security analysis on android: A systematic literature review. *IEEE Access*, 12:57261–57287, 2024.

[15] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Automatic reverse engineering of malware emulators. In *2009 30th IEEE Symposium on Security and Privacy*, pages 94–109, 2009.

[16] L. Dresel, M. Protsenko, and T. Müller. Artist: The android runtime instrumentation toolkit. In *2016 11th International Conference on Availability, Reliability and Security (ARES)*, pages 107–116, 2016.

[17] M. Steinböck, J. Troost, W. van Beijnum, J. Seredynski, H. Bos, M. Lindorfer, and A. Continella. SoK: Hardening Techniques in the Mobile Ecosystem - Are We There Yet? In *Proceedings of the 10th IEEE European Symposium on Security and Privacy (EuroS&P)*, 2025.

[18] Y. Sharma, D. Tomar, R.K Pateriya, and S. Bhandari. Mosdroid: Obfuscation-resilient android malware detection using multisets of encoded opcode sequences. *Computers & Security*, 152:104379, 2025.

[19] H. Aghakhani, F. Gritti, F. Mecca, M. Lindorfer, S. Ortolani, D. Balzarotti, G. Vigna, and C. Kruegel. When Malware is Packin' Heat; Limits of Machine Learning Classifiers Based on Static Analysis Features. In *Proceedings of the 27th Network and Distributed System Security Symposium (NDSS)*, 2020.

[20] J. Bleier and M. Lindorfer. Of ahead time: Evaluating disassembly of android apps compiled to binary oats through the art. In *Proceedings of the 16th European Workshop on System Security*, EUROSEC '23, page 21–29, New York, NY, USA, 2023. Association for Computing Machinery.

[21] R. Guo, Q. Liu, M. Zhang, N. Hu, and H. Lu. A survey of obfuscation and deobfuscation techniques in android code protection. In *2022 7th IEEE International Conference on Data Science in Cyberspace (DSC)*, pages 40–47, 2022.

[22] U. Kargén, N. Mauthe, and N. Shahmehri. Characterizing the use of code obfuscation in malicious and benign android apps. In *Proceedings of the 18th International Conference on Availability, Reliability and Security*, pages 1–12. ACM, 8 2023.

[23] Jakob Bleier and Martina Lindorfer. Back to the Binary: Revisiting Similarities of Android Apps. *Phrack*, 72, 2025.

[24] A. Niroshan, S. Seneviratne, and A. Seneviratne. *State of Obfuscation: A Longitudinal Study of Code Obfuscation Practices in Google Play Store*, page 592–596. Association for Computing Machinery, New York, NY, USA, 2025.

[25] B. Molina-Coronado, A. Ruggia, U. Mori, A. Merlo, A. Mendiburu, and J. Miguel-Alonso. Light up that droid! on the effectiveness of static analysis features against app obfuscation for android malware detection. *J. Netw. Comput. Appl.*, 235(C), March 2025.

[26] M. Wong and D. Lie. Tackling runtime-based obfuscation in android with TIRO. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1247–1262, Baltimore, MD, August 2018. USENIX Association.

[27] J. Shu, J. Li, Y. Zhang, and D. Gu. Android app protection via interpretation obfuscation. In *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, pages 63–68. IEEE, 8 2014.

[28] Y. Zhao, Z. Tang, G. Ye, D. Peng, D. Fang, X. Chen, and Z. Wang. Compile-time code virtualization for android applications. *Computers & Security*, 94:101821, 2020.

[29] N. Zhang, D. Xu, J. Ming, J. Xu, and Q. Yu. Inspecting virtual machine diversification inside virtualization obfuscation. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 3051–3069, 2025.

[30] Z. He, G. Ye, L. Yuan, Z. Tang, X. Wang, J. Ren, W. Wang, J. Yang, D. Fang, and Z. Wang. Exploiting binary-level code virtualization to protect android applications against app repackaging. *IEEE Access*, 7:115062–115074, 2019.

[31] D. Quarta, M. Ianni, A. Machiry, Y. Fratantonio, E. Gustafson, D. Balzarotti, M. Lindorfer, G. Vigna, and C. Kruegel. Tarnhelm: Isolated, Transparent & Confidential Execution of Arbitrary Code in ARM's TrustZone. In *Proceedings of the 1st Workshop on Research on offensive and defensive techniques in the Context of Man At The End Attacks (CheckMATE)*, 2021.

[32] L. Xue, Y. Yan, L. Yan, M. Jiang, X. Luo, D. Wu, and Y. Zhou. Parema: an unpacking framework for demystifying vm-based android packers. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 152–164. ACM, 7 2021.

[33] P. Graux, J. Lalande, P. Wilke, and V. Tong. Abusing android runtime for application obfuscation. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 616–624. IEEE, 9 2020.

[34] G. You, G. Kim, S. Han, M. Park, and S. Cho. Deoptfuscator: Defeating advanced control-flow obfuscation using android runtime (art). *IEEE Access*, 10:61426–61440, 2022.

[35] J. Bleier, F. Kehrer, J. Cito, and M. Lindorfer. Profile Coverage: Using Android Compilation Profiles to Evaluate Dynamic Testing. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2025.

[36] X. Zhang, F. Breitinger, E. Luechinger, and S. O'Shaughnessy. Android application forensics: A survey of obfuscation, obfuscation detection and deobfuscation techniques and their impact on investigations. *Forensic Science International: Digital Investigation*, 39:301285, 2021.

[37] Z. Wang, Y. Shan, Z. Yang, R. Wang, and S. Song. Semantic redirection obfuscation: A control flow obfuscation based on android runtime. In *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 1756–1763. IEEE, 12 2020.

[38] V. Balachandran, Sufatrio, D. Tan, and V. Thing. Control flow obfuscation for android applications. *Computers and Security*, 61:72–93, 2016.

[39] L. Zobernig, S. Galbraith, and G. Russello. When are opaque predicates useful? In *18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications*, pages 168–175, 2019.

[40] T. László and Á. Kiss. Obfuscating c++ programs via control flow flattening. *Conference: 10th Symposium on Programming Languages and Software Tools (SPLST 2007)*, 30(1):3–19, 6 2007.

[41] S. Ilić and S. Đukić. Protection of android applications from decompilation using class encryption and native code. In *2016 Zooming Innovation in Consumer Electronics International Conference (ZINC)*, pages 10–11, 2016.

[42] D. Maiorca, D. Ariu, I. Corona, M. Aresu, and G. Giacinto. Stealth attacks: An extended insight into the obfuscation effects on android malware. *Comput. Secur.*, 51(C):16–31, 6 2015.

[43] V. Haupert, D. Maier, N. Schneider, J. Kirsch, and T. Müller. Honey, i shrunk your app security: The state of android app hardening. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 69–91. Springer International Publishing, 06 2018.

[44] Y. Jing, Z. Zhao, G. Ahn, and H. Hu. Morpheus: Automatically generating heuristics to detect android emulators. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, page 216–225, New York, NY, USA, 2014. Association for Computing Machinery.

[45] S. Li, R. Li, S. Yang, and W. Diao. Android's cat-and-mouse game: Understanding evasion techniques against dynamic analysis. In *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*, pages 192–203, 2024.

[46] S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang, and K. Zhang. Understanding android obfuscation techniques: A large-scale investigation in the wild. *CoRR*, abs/1801.01633, 2018.

[47] R. Rolles. Unpacking virtualization obfuscators. In *Proceedings of the 3rd USENIX Conference on Offensive Technologies*, WOOT'09, page 1, USA, 2009. USENIX Association.

[48] J. Kinder. Towards static analysis of virtualization-obfuscated binaries. In *2012 19th Working Conference on Reverse Engineering*, pages 61–70, 2012.

[49] K. Coogan, G. Lu, and S. Debray. Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, page 275–284, New York, NY, USA, 2011. Association for Computing Machinery.

[50] J. Levin. *Android Internals::Developer's View*. Technologeeks.com, 2023. ISBN: 978-0-9910555-1-9.

[51] G. Kc, A. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, page 272–280, New York, NY, USA, 2003. Association for Computing Machinery.

[52] D. Knuth. *The art of computer programming: Seminumerical algorithms, volume 2*. Addison-Wesley Professional, 2014.

Online References

[53] StatCounter Global Stats. *Mobile Operating System Market Share Worldwide*. https://gs.statcounter.com/os-market-share/mobile/worldwide (Accessed: 2026-01-31).

[54] Guardsquare NV. *ProGuard*. https://github.com/Guardsquare/proguard (Accessed: 2026-01-31).

[55] Google LLC. *Enable app optimization (R8)*. https://developer.android.com/topic/performance/app-optimization/enable-app-optimization (Accessed: 2026-01-31).

[56] P. Faruki, H. Fereidooni, V. Laxmi, M. Conti, and M. Gaur. *Android Code Protection via Obfuscation Techniques: Past, Present and Future Directions*. https://arxiv.org/abs/1611.10231 (Accessed: 2026-01-31).

[57] Guardsquare NV. *DexGuard introduces code virtualization for Android apps*. https://www.guardsquare.com/blog/dexguard-introduces-code-virtualization-android (Accessed: 2026-01-31).

[58] Google LLC. *Android*. https://www.android.com/ (Accessed: 2026-01-31).

[59] Google LLC. *Open Hands Alliance*. https://www.openhandsetalliance.com/android_overview.html (Accessed: 2026-01-31).

[60] Google LLC. *Android Open Source Project*. https://source.android.com/ (Accessed: 2026-01-31).

[61] Google LLC. *Android Compatibility Definition Document*. https://source.android.com/docs/compatibility/cdd (Accessed: 2026-01-31).

[62] Google LLC. *Google Play Store*. https://play.google.com/store/apps (Accessed: 2026-01-31).

[63] Google LLC. *Android architecture*. https://developer.android.com/guide/platform (Accessed: 2026-01-31).

[64] Google LLC. *Android kernel overview*. https://source.android.com/docs/core/architecture/kernel (Accessed: 2026-01-31).

[65] L. Torvald. *Android kernel overview*. https://github.com/torvalds/linux (Accessed: 2026-01-31).

[66] Google LLC. *System and kernel security*. https://source.android.com/docs/security/overview/kernel-security (Accessed: 2026-01-31).

[67] Google LLC. *Hardware abstraction layer*. https://source.android.com/docs/core/architecture/hal (Accessed: 2026-01-31).

[68] Google LLC. *Android Camera module*. https://source.android.com/docs/core/camera (Accessed: 2026-01-31).

[69] Google LLC. *Android Zygote*. https://source.android.com/docs/core/runtime/zygote (Accessed: 2026-01-31).

[70] Google LLC. *SurfaceFlinger and WindowManager*. https://source.android.com/docs/core/graphics/surfaceflinger-windowmanager (Accessed: 2026-01-31).

[71] Google LLC. *AOSP architecture*. https://source.android.com/docs/core/architecture (Accessed: 2026-01-31).

[72] Google LLC. *Android runtime and Dalvik*. https://source.android.com/docs/core/runtime (Accessed: 2026-01-31).

[73] Google LLC. *Apache Harmony - Open Source Java Platform*. https://harmony.apache.org/ (Accessed: 2026-01-31).

[74] Google LLC. *Android API reference*. https://developer.android.com/reference (Accessed: 2026-01-31).

[75] Google LLC. *Android 5.0 Compatibility Definition*. https://source.android.com/docs/compatibility/5.0/android-5.0-cdd (Accessed: 2026-01-31).

[76] B. Cheng and B. Buzbee (Google). *A JIT Compiler for Android's Dalvik VM*. https://web.archive.org/web/20150714081347/http://www.android-app-developer.co.uk/android-app-development-docs/android-jit-compiler-androids-dalvik-vm.pdf (Accessed: 2026-01-31).

[77] Google LLC. *Implement ART just-in-time compiler*. https://source.android.com/docs/core/runtime/jit-compiler (Accessed: 2026-01-31).

[78] Google LLC. *Verifying app behavior on the Android runtime (ART)*. https://developer.android.com/guide/practices/verifying-apps-art (Accessed: 2026-01-31).

[79] Google LLC. *Android ART Configuration and Compilation Options*. https://source.android.com/docs/core/runtime/configure (Accessed: 2026-01-31).

[80] Google LLC. *Android 8.0 ART improvements*. https://source.android.com/docs/core/runtime/improvements (Accessed: 2026-01-31).

[81] Google LLC. *Android Application fundamentals*. https://developer.android.com/guide/components/fundamentals (Accessed: 2026-01-31).

[82] Google LLC. *Privileged Permission Allowlist*. https://source.android.com/docs/core/permissions/perms-allowlist (Accessed: 2026-01-31).

[83] Google LLC. *Scoped Storage*. https://source.android.com/docs/core/storage/scoped (Accessed: 2026-01-31).

[84] Google LLC. *Android App Bundle FAQ*. https://developer.android.com/guide/app-bundle/faq (Accessed: 2026-01-31).

[85] Google LLC. *Android Studio Projects overview*. https://developer.android.com/studio/projects (Accessed: 2026-01-31).

[86] Google LLC. *Support 64-bit architectures*. https://developer.android.com/google/play/requirements/64-bit (Accessed: 2026-01-31).

[87] Google LLC. *Android Application Signing*. https://source.android.com/docs/security/features/apksigning (Accessed: 2026-01-31).

[88] Google LLC. *APK signature scheme v2*. https://source.android.com/docs/security/features/apksigning/v2 (Accessed: 2026-01-31).

[89] Google LLC. *Dalvik executable format*. https://source.android.com/docs/core/runtime/dex-format (Accessed: 2026-01-31).

[90] Google LLC. *Dalvik bytecode format*. https://source.android.com/docs/core/runtime/dalvik-bytecode (Accessed: 2026-01-31).

[91] Google LLC. *Android Native Development Kit*. https://developer.android.com/ndk/guides (Accessed: 2026-01-31).

[92] Oracle. *Java Native Interface Documentation*. https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html (Accessed: 2026-01-31).

[93] Google LLC. *Security-Enhanced Linux in Android*. https://source.android.com/docs/security/features/selinux (Accessed: 2026-01-31).

[94] Google LLC. *Application Sandbox*. https://source.android.com/docs/security/app-sandbox (Accessed: 2026-01-31).

[95] Google LLC. *Android Encryption*. https://source.android.com/docs/security/features/encryption (Accessed: 2026-01-31).

[96] Google LLC. *Android Verified Boot*. https://source.android.com/docs/security/features/verifiedboot (Accessed: 2026-01-31).

[97] Google LLC. *Trusted Execution Environment*. https://source.android.com/docs/security/features/trusty (Accessed: 2026-01-31).

[98] Google LLC. *Android Keystore system*. https://developer.android.com/privacy-and-security/keystore (Accessed: 2026-01-31).

[99] Google LLC. *Android Biometric Authentication*. https://developer.android.com/identity/sign-in/biometric-auth (Accessed: 2026-01-31).

[100] Google LLC. *Android 4.2 Compatibility Definition*. https://source.android.com/docs/compatibility/4.2/android-4.2-cdd (Accessed: 2026-01-31).

[101] Google LLC. *Permissions on Android*. https://developer.android.com/guide/topics/permissions/overview (Accessed: 2026-01-31).

[102] Google LLC. *Android Package Manager*. https://developer.android.com/reference/android/content/pm/PackageManager (Accessed: 2026-01-31).

[103] Google LLC. *About Android App Bundles*. https://developer.android.com/guide/app-bundle (Accessed: 2026-01-31).

[104] JesusFreke / Google. *smali/baksmali*. https://github.com/google/smali (Accessed: 2026-01-31).

[105] Google LLC. *Android Application Signing Publish*. https://developer.android.com/studio/publish/app-signing (Accessed: 2026-01-31).

[106] C. Tumbleson. *Apktool - A tool for reverse engineering Android apk files*. https://apktool.org (Accessed: 2026-01-31).

[107] LLDB Team. *Low Level Debugger (LLDB)*. https://lldb.llvm.org/ (Accessed: 2026-01-31).

[108] Free Software Foundation. *GNU Debugger (GDB)*. https://www.sourceware.org/gdb/ (Accessed: 2026-01-31).

[109] Bellard et al. *QEMU*. https://www.qemu.org/ (Accessed: 2026-01-31).

[110] Google LLC. *Run apps on the Android Emulator*. https://developer.android.com/studio/run/emulator (Accessed: 2026-01-31).

[111] O. Ravnås. *Frida - Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers*. https://frida.re (Accessed: 2026-01-31).

[112] O. Ravnås. *Frida - Modes of Operation*. https://frida.re/docs/modes/ (Accessed: 2026-01-31).

[113] O. Ravnås. *Frida - Android Example*. https://frida.re/docs/examples/android/ (Accessed: 2026-01-31).

[114] skylot. *jadx - Dex to Java decompiler*. https://github.com/skylot/jadx (Accessed: 2026-01-31).

[115] National Security Agency (NSA). *Ghidra*. https://github.com/NationalSecurityAgency/ghidra (Accessed: 2026-01-31).

[116] VMProtect Software. *VMProtect*. https://vmpsoft.com/vmprotect/user-manual (Accessed: 2026-01-31).

[117] C. Collberg. *Tigress*. https://tigress.wtf/virtualize.html (Accessed: 2026-01-31).

[118] OWASP. *OWASP Mobile Top 10*. https://owasp.org/www-project-mobile-top-10 (Accessed: 2026-01-31).

[119] Google LLC. *Android App Minification*. https://developer.android.com/build/shrink-code (Accessed: 2026-01-31).

[120] Google LLC. *Android application manifest overview*. https://developer.android.com/guide/topics/manifest/manifest-intro (Accessed: 2026-01-31).

[121] Google LLC. *Android 8.0 Compatibility Definition*. https://source.android.com/docs/compatibility/8.0/android-8.0-cdd (Accessed: 2026-01-31).

[122] Google LLC. *Android Debug*. https://developer.android.com/reference/android/os/Debug (Accessed: 2026-01-31).

[123] M. Kerrisk. *ptrace(2) — Linux manual page*. https://man7.org/linux/man-pages/man2/ptrace.2.html (Accessed: 2026-01-31).

[124] LLC SaurikIT. *Cydia Substrate*. https://www.cydiasubstrate.com (Accessed: 2026-01-31).

[125] Chainfire. *SuperSU*. https://supersuroot.org/ (Accessed: 2026-01-31).

[126] J. Wu. *Magisk*. https://github.com/topjohnwu/Magisk (Accessed: 2026-01-31).

[127] Google LLC. *Google Play Integrity API*. https://developer.android.com/google/play/integrity/overview (Accessed: 2026-01-31).

[128] Oracle (Glen McCluskey). *Using Java Reflection*. https://www.oracle.com/technical-resources/articles/java/javareflection.html (Accessed: 2026-01-31).

[129] Oracle. *Java Reflection API*. https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/package-summary.html (Accessed: 2026-01-31).

[130] Google LLC. *Restrictions on non-SDK interfaces*. https://developer.android.com/guide/app-compatibility/restrictions-non-sdk-interfaces (Accessed: 2026-01-31).

[131] Google LLC. *Android DexClassLoader*. https://developer.android.com/reference/dalvik/system/DexClassLoader (Accessed: 2026-01-31).

[132] Google LLC. *Android PathClassLoader*. https://developer.android.com/reference/dalvik/system/PathClassLoader (Accessed: 2026-01-31).

[133] Google LLC. *Android InMemoryDexClassLoader*. https://developer.android.com/reference/dalvik/system/InMemoryDexClassLoader (Accessed: 2026-01-31).

[134] Oreans Technologies. *CodeVirtualizer*. https://www.oreans.com/CodeVirtualizer.php (Accessed: 2026-01-31).

[135] Google LLC. *Android Open Source Project - Android 15 (r14)*. https://cs.android.com/android/platform/superproject/+/android-15.0.0_r14: (Accessed: 2026-01-31).

[136] Google LLC. *Try Android development*. https://source.android.com/docs/setup/start (Accessed: 2026-01-31).

[137] Google LLC. *Android Open Source Project - Android 11 (r40) mterp*. https://cs.android.com/android/platform/superproject/+/android-11.0.0_r40:art/runtime/interpreter/mterp/ (Accessed: 2026-01-31).

[138] Google LLC. *Android Opcodes*. https://developer.android.com/reference/dalvik/bytecode/Opcodes (Accessed: 2026-01-31).

[139] J. Carolus. *Chess*. https://f-droid.org/de/packages/jwtc.android.chess/ (Accessed: 2026-01-31).

[140] Nononsense Apps. *Feeder*. https://f-droid.org/de/packages/com.nononsenseapps.feeder/ (Accessed: 2026-01-31).

[141] Fossify. *Fossify Calculator Beta*. https://f-droid.org/en/packages/org.fossify.math/ (Accessed: 2026-01-31).

[142] Fossify. *Fossify Calendar*. https://f-droid.org/packages/org.fossify.calendar/ (Accessed: 2026-01-31).

[143] Fossify. *Fossify Clock Beta*. https://f-droid.org/en/packages/org.fossify.clock/ (Accessed: 2026-01-31).

[144] Fossify. *Fossify Contacts*. https://f-droid.org/en/packages/org.fossify.contacts/ (Accessed: 2026-01-31).

[145] Fossify. *Fossify File Manager*. https://f-droid.org/en/packages/org.fossify.filemanager/ (Accessed: 2026-01-31).

[146] Fossify. *Fossify Gallery*. https://f-droid.org/en/packages/org.fossify.gallery/ (Accessed: 2026-01-31).

[147] Fossify. *Fossify Home*. https://f-droid.org/de/packages/org.fossify.home/ (Accessed: 2026-01-31).

[148] Fossify. *Fossify Messages*. https://f-droid.org/en/packages/org.fossify.messages/ (Accessed: 2026-01-31).

[149] Fossify. *Fossify Music Player*. https://f-droid.org/de/packages/org.fossify.musicplayer/ (Accessed: 2026-01-31).

[150] Fossify. *Fossify Notes*. https://f-droid.org/de/packages/org.fossify.notes/ (Accessed: 2026-01-31).

[151] Fossify. *Fossify Paint*. https://f-droid.org/de/packages/org.fossify.paint/ (Accessed: 2026-01-31).

[152] W. Heller. *FREE Browser*. https://f-droid.org/de/packages/org.woheller69.browser/ (Accessed: 2026-01-31).

[153] I. Calì. *OpenMoneyBox*. https://f-droid.org/de/packages/com.igisw.openmoneybox/ (Accessed: 2026-01-31).

[154] T. Williamson. *QuickWeather*. https://f-droid.org/en/packages/com.ominous.quickweather/ (Accessed: 2026-01-31).

[155] Trianguloy. *Simple Clipboard Editor*. https://f-droid.org/de/packages/com.trianguloy.clipboardeditor/ (Accessed: 2026-01-31).

[156] SECUSO. *Sudoku Privacy Friendly*. https://f-droid.org/de/packages/org.secuso.privacyfriendlysudoku/ (Accessed: 2026-01-31).

[157] E. Messulam. *TicTacToe Game*. https://f-droid.org/packages/com.emmanuelmess.tictactoe/ (Accessed: 2026-01-31).

[158] C. Gultnieks et al. *F-Droid*. https://f-droid.org/en/ (Accessed: 2026-01-31).

[159] Google LLC. *UI/Application Exerciser Monkey*. https://developer.android.com/studio/test/other-testing-tools/monkey (Accessed: 2026-01-31).

[160] PassMark Software. *PassMark PerformanceTest*. https://play.google.com/store/apps/details?id=com.passmark.pt_mobile (Accessed: 2026-01-31).

[161] Georgiu C. *Obfuscapk*. https://github.com/ClaudiuGeorgiu/Obfuscapk (Accessed: 2026-01-31).