UNIVERSITY OF GENOA

PH.D. PROGRAM IN
SECURITY, RISK, AND VULNERABILITY
XXXVI CYCLE

Curriculum: *Cybersecurity and Reliable Artificial Intelligence*

Shades of Gray: Delving Surreptitious Code in Android Applications

by

Antonio Ruggia

January 26, 2024

Supervisor Prof. Alessio Merlo, *CASD - School of Advanced Defense Studies*

Ph.D. Coordinator Prof. Alessandro Armando, *University of Genoa*

Ph.D. Curriculum Coordinator Prof. Luca Oneto, *University of Genoa*

Ext. Reviewers Prof. Lorenzo Cavallaro, *University College of London (UCL)*
Dr. Daniele Cono D'Elia, *Sapienza University of Rome*

Dibris

Department of Informatics, Bioengineering, Robotics and Systems Engineering

Abstract

In the last decade, we have faced the rise of mobile devices as a fundamental tool in our everyday lives. At the time of writing, there are more than 4.2 billion active mobile users, and 71% of them take advantage of the Android operating system. The functionalities of smartphones are enriched by mobile applications through which users can perform every operation that in the past has been made possible only on computers or web applications. Although this introduces advantages and conveniences for the end users as they can perform operations from the comfort of their mobile devices, it also presents several security challenges. In particular, Android devices and applications become a desirable target for large-scale malware distribution. Thus, malicious applications have constantly evolved, becoming increasingly sophisticated and stealthy. The aim of malware authors is twofold. They aim at fooling as many final users as possible to install their malicious applications *and* they pay special attention to flying under the radar of analysis tools to avoid manual and automatic detection to operate undisturbed. The response from the community was swift, and many researchers have ventured to defend this system, constantly proposing protection methodologies or novel and more robust analysis techniques.

This thesis aims to contribute to this cat-and-mouse game by analyzing some of the open problems affecting different security aspects of the Android ecosystem. This thesis starts with analyzing modern repackaging attacks – widely used by malicious actors to distribute malware samples stealthily – and introduces a novel anti-repackaging scheme that considers the security requirements a protection technique should follow.

Then, this thesis focuses on the security analysis of modern anti-analysis and protection techniques exploited by Android applications. In particular, it focuses on revealing the role of native (C/C++) code in malware samples and the evasive techniques used by both benign and malicious applications. To those aims, this thesis also proposes a novel methodology to reverse engineering Android applications focusing on suspicious patterns related to native components and a probe-based sandbox – a dynamic analysis system – which circumvents evasive techniques thanks to a substantial engineering effort, making the applications under analysis believe they are running on an actual device. Our results depict typical behaviors

of modern malware and its evolution and reveal insights about the anti-analysis techniques' purpose, differences, and relationships between legitimate and harmful behaviors.

Finally, the analysis of the native layer brought to light a novel state inference attack that can be abused by modern malware to carry out more complex attacks, such as phishing. This thesis also highlights how there still needs to be a system in place to allow applications to protect themselves against this novel attack vector.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With over 3 billion active users and a market share covering almost 71% of smartphones, Android is today one of the most widely used operating systems in the world [205, 206]. Most of this success is due to two different aspects. The first one is related to the open-source nature that allows different vendors to produce devices based on Android. The second one is the more straightforward application development process compared to other mobile operating systems: Google currently offers a plethora of platforms that support the developers in designing, implementing, testing, and sharing their applications [100]. This success can also be seen in how many products and services are now available in the several app stores (e.g., Google Play Store, Samsung App Store), which vary from email clients to banking applications. All this allows the user to perform more operations from the comfort of their smartphones.

From a security standpoint, however, this high market share makes Android a desirable target for malware authors, allowing them to reach many mobile users. Since August 2010, when security companies detected the first Android malware [121], Android has attracted more and more attackers. In addition, attack vectors previously used only to compromise other platforms (e.g., computers and web applications) have also been adapted to the mobile ecosystem, creating new types and categories of attacks that are Android-specific [129]. Phishing attacks are a primary example. In a standard web-based phishing attack, a malicious actor aims to trick users into doing "the wrong thing", such as clicking a bad link in an email that will download the malware or re-direct them to a phony website. Conversely, as a first step in the Android phishing attack, an attacker must install the malicious application on the victim's device. Then, when the user opens the target application, the malicious one (identical to the original) pops up on top of the UI instead of the original application. However, installing the malware is insufficient for an attacker to access and exfiltrate sensitive

data: the malicious actor has to find a way to infer when the user interacts with the target application to understand the exact moment she can carry out the overlay attack.

The spread of these attacks has pushed researchers to study new countermeasures to improve the security and privacy of users, devices, and applications. In contrast, malicious samples have constantly evolved, reacting to these remediations and becoming increasingly sophisticated to fool the final users into installing and using them. Thus, malware authors and security researchers engaged in a constant arms race, of which *app repackaging* is a clear example (recent malicious repackaged examples are [233, 193, 197, 59]). App repackaging refers to the process of customizing an existing mobile application and re-distributing it in the wild to deceive the final user into installing the repackaged sample instead of the original one [149]. In this scenario, since an Android application is publicly available in an app store, an attacker can easily retrieve the app bundle (i.e., the APK file, which is an archive containing the application's content), reverse engineer it, inject some malicious code, and re-distribute a modified version of the original sample. Thus, an attacker does not have to develop an application from scratch (she only generates the malicious payload), and the repackaged sample inherits the widespread market share and the popularity of the original legitimate application. To protect from this trend, modern Android samples employ repackaging protection techniques (also known as *anti-repackaging*), which insert proper integrity controls into the application such that a repackaged version of that sample would not work correctly. However, a novel and growing trend in repackaging attacks exploits the Android virtualization technique [50], in which malicious code can run together with the victim application in a virtual container. In such a scenario, the attacker can directly build a malicious container capable of hosting the victim sample instead of tampering with it, avoiding being detected by the current anti-repackaging schemes.

Alongside the repackaging threat, modern malicious samples must fly under the radar of analysis tools, remaining undetected to operate undisturbed. To this end, malware authors developed new techniques to make static analysis challenging (e.g., obfuscation) or even impossible (e.g., removing some code portions and loading them at runtime). For instance, while allowing native (C/C++) code in Android applications has a strong positive impact from a performance perspective, it dramatically complicates the analysis of a sample because bytecode and native code need different abstractions and analysis algorithms, and they thus pose other challenges and limitations. Moreover, malware authors try to avoid dynamic analysis by putting in place *evasive* controls to understand whether their applications are under analysis. However, all these techniques are also adopted by benign samples to make reverse engineering harder and protect themselves from a plethora of attacks (e.g., repackaging).

Every program that contains both a secret – e.g., the intellectual property for goodware and the malicious payload for malware – and any technique for preventing an attack against this secret (e.g., evasive controls) is known as *surreptitious software* [153]. Surreptitious code is part of a middle "gray area" between malware and goodware: anti-analysis techniques are typical behavior of malicious Android applications, whose goal is to remain undetected by users and security software to perform harmful activities covertly. On the other hand, goodware could also adopt surreptitious code implementing anti-analysis techniques to protect, for instance, users' privacy and data security or the developers' intellectual property. Thus, such techniques do not necessarily imply maliciousness: they are typically concerned with preventing others from exploiting the intellectual effort invested in producing a piece of software, regardless of whether the effort is for uncovering malicious purposes.

This middle layer is crucial in the cat-and-mouse game between malware and goodware. To avoid detection, malware tends to exhibit behavior more and more similar to goodware, which, in turn, has adopted anti-analysis and protection techniques typically developed and used by malicious authors. The scientific literature has also proposed many novel anti-analysis techniques and investigated their impact, but some critical aspect still needs to be studied. In particular, differentiating and correctly investigating how samples (ab)use surreptitious code could shed light on their real nature, such as in a malware classification task scenario. Moreover, modern benign applications often leverage ineffective protection techniques, making them easy to attack by malicious actors. For instance, state-of-the-art anti-repackaging protections are inefficient against specific attack scenarios (e.g., virtualization-based attacks); thus, an attacker can easily bypass and deactivate the protection mechanisms. The goal of benign actors is not only to implement protection techniques but also to ensure adequate controls. In fact, according to the OWASP Mobile Top Ten, one of the main issues in securing Android applications is improper and insufficient binary protections [164].

In conclusion, shedding light on such aspects is essential because it allows malware analysts and industry experts to improve both malware understanding and the protection of benign samples, making the Android world safer for its billions of users.

1.1 Research Contributions

The research presented in this thesis has been guided by several key questions relevant to understanding the usage or improving the effectiveness of surreptitious code in Android applications. For instance, anti-analysis techniques sit in the gray area – a no-man's land – between malicious code and legitimate conventional software. Such controls are heavily

adopted by both malware and goodware, but their use is for intrinsically different purposes. On one hand, malware samples aim to avoid detection by flying under the radar of static and dynamic analysis tools. At the same time, benign applications protect their code from reverse engineering and specific client-side attacks or ensure that the users' sensitive data (e.g., bank access tokens) are not stored, for instance, in a rooted device [149, 195, 29].

Over the years, researchers studied anti-analysis and protection mechanisms from both the benign and malicious points of view (e.g., [29, 149, 57, 151, 27, 4]); however, nowadays, a comprehensive analysis and categorization of them is still needed. Thus, this thesis started by collecting and categorizing all documented techniques for protecting Android samples – benign and malicious – from static and dynamic analysis techniques. In particular, it focuses on introducing a comprehensive per-purpose *taxonomy of evasive controls* – i.e., application techniques to prevent the runtime inspection and analysis of their behavior – for the Android ecosystem. However, this categorization is just the first step: my research has also looked to understand and measure their different usage and security implications. Alongside the measurements, this thesis documents a significant engineering effort to support Android security experts in their work and develop new security techniques or make the existing ones more secure, effective, and practical.

In the following paragraphs, I briefly illustrate the principal contributions of this thesis.

In the first part, I explored how to make Android applications safer, raising the bar for attackers who want to compromise them through a repackaging attack. Android app repackaging refers to a man-at-the-end attack in which a malicious user analyses and modifies the content of the APK file to redistribute a modified version of the original application [149]. Thus, anti-repackaging techniques try counteracting this attack by adding logical controls at compile-time (i.e., in this form of surreptitious code, the secret to protect is the applications' code from being tampered with). However, such protection schemes are ineffective against novel attack vectors: repackaging attacks through app-level virtualization. Only two solutions have been specifically designed to address virtualization-based repackaging attacks. Still, their effectiveness could be improved since they rely on static taint analysis, thus unable to evaluate code dynamically loaded at runtime. Hence, this thesis proposes *MARVEL*, a *virtualization-based anti-repackaging scheme* that limits both standard and virtualization-based attacks. *MARVEL* deploys a mutual integrity verification mechanism between the trusted container and the plugin and strongly relies on the virtualization technique to build a secure virtual environment where only protected applications can run and be checked at runtime. Moreover, this thesis also demonstrated the strengths of the proposed methodology against the repackaging threat compared to the other state-of-the-art solutions.

In the following parts, this thesis investigates several aspects of surreptitious code in malicious and benign samples and highlights their different management and purpose.

First, this thesis proposes a novel methodology to reverse engineering Android applications focusing on suspicious patterns related to *native components*. Briefly, Android applications are written in high-level languages (Java and Kotlin), but thanks to the Java Native Interface (JNI), Android also supports calling native (C/C++) library functions. Contrary to Android high-level languages, native code is architecture-dependent, and every application has to ship an ELF file (in the form of a Shared Object) for each architecture it wants to support. Moreover, JNI serves various purposes; for instance, it can interact with the Android RunTime (ART), instantiate new objects, and perform low-level operations. This versatility and the additional complexity of its analysis have led malware authors to increasingly use the native layer to hide malicious code and perform suspicious (or even malicious) operations. For instance, if malware authors knew that specific antivirus solutions support only x86-based architectures, they could limit the malicious logic to ARM and insert a harmless code into the x86 library. Thus, the sandbox only loads the x86 library, and the malicious component would avoid examination. Still, the malware would behave as intended (maliciousness) when run on an actual ARM-based device. Moreover, this chapter shows how the static module of many Android anti-malware engines ignores the presence of native components or performs only straightforward controls, highlighting the lack of knowledge of these engines on how malware exploits native code. Thus, to improve the understanding of the native layer, we performed the first longitudinal analysis of the native code in Android malware over the past ten years and compared the recent malicious samples with actual top benign applications on the Google Play Store. The results depict typical behaviors of modern malware, its evolution, and how it (ab)uses the native layer to complicate the analysis, especially with dynamic code loading and novel anti-analysis techniques.

Second, the native layer analysis contributed to the discovery of a novel and undocumented *state inference attack* on Android that enables a malware author to build more complex attacks (e.g., a phishing one) from her malicious application installed on the victim's smartphone. In detail, the attacker abuses the *inotify* API, a mechanism for monitoring file system events, to infer the application opened by the user. State inference attacks are the basic block of the phishing one because they allow an attacker to infer the exact time to perform the phishing attack through, for instance, an overlay technique. Moreover, phishing attacks are particularly problematic for mobile platforms because Android does not provide enough information for a user to reliably distinguish a legitimate application from a malicious one that spoofs the user interface. Thus, this part of the thesis studied the impact of

such a new attack concerning different Android versions in which the malware runs and the permissions granted to the malicious sample at runtime. It is worth noticing that this vulnerability has existed but has remained hidden for several years. The most disturbing finding is that if the attacker knows the installation path of the target application, *all* Android applications are vulnerable, regardless of the system version. At the time of writing, all Android versions and devices are vulnerable to this attack. Getting the installation path is, nowadays, a capability that is only protected by a normal permission, and to make matters worse, there are workarounds to get it even without such permission. Moreover, the analysis demonstrated that this attack is not limited to only the standard Android operating system: every Android-based custom ROM tested is vulnerable.

Finally, while the scientific literature has proposed many *evasive techniques* and investigated their impact, one aspect still needs to be studied: how and to what extent Android applications, both malware and goodware, use such controls. Thus, as a last contribution, this thesis fills this gap. A dynamic analysis system (known as *sandbox*) is needed to execute and analyze the samples. The term "sandbox" in the context of malware analysis is typically used to describe a dynamic analysis tool that operates in a secure environment for analyzing and monitoring potentially harmful code without endangering the host or the network. Given that state-of-the-art sandboxes do not make such an analysis possible, this thesis proposed *DroidDungeon*, a novel probe-based sandbox, which circumvents evasive techniques thanks to a substantial engineering effort, making the applications under analysis believe they are running on an actual device. To the best of our knowledge, currently *DroidDungeon* is the only solution providing anti-evasion capabilities, maintainability, and scalability at once. Using this sandbox, this chapter studies evasive controls in both benign and malicious Android applications, revealing insights about their purpose and differences. It also analyzes how the execution of an application differs depending on the presence or absence of evasive countermeasures, highlighting the types of operations that malware typically hides under evasive behaviors. The main results is that 14% and 4% of malicious and benign samples refrain from running in an analysis environment that does not correctly mitigate evasive controls. This proves once and for all that, today, it is necessary to consider Android-specific surreptitious code and these gray area-related aspects to improve the malware detection and classification task.

1.2 Thesis outline

The thesis consists of a total of 7 chapters.

Chapter 2 presents the necessary background knowledge on Android to understand the main contributions of the thesis. Moreover, it proposes a comprehensive taxonomy of all the documented techniques for protecting Android applications, dividing them based on their objective. Even if this part discusses a new per-purpose categorization of evasive controls, it might be seen as part of the common background of the thesis as it considers only well-known anti-analysis techniques.

Chapter 3 investigates the virtualization-based repackaging threat and introduces MARVEL, the first anti-repackaging scheme able to react to this problem dynamically. It is based on the paper "Repack Me If You Can: An Anti-Repackaging Solution Based on Android Virtualization" published at the Annual Computer Security Applications Conference (ACSAC) 2021, in joint work with Alessio Merlo and Luca Verderame of the University of Genoa, and Eleonora Losiouk and Mauro Conti of the University of Padua [183]. The MARVEL implementation has been submitted for the artifact evaluation of the conference; I received the REUSABLE badge, and MARVEL won the best artifact award.

In Chapter 4, this thesis introduces a comprehensive methodology to analyze all aspects of the surreptitious native code in Android applications. Moreover, as a use case, it discusses the impact of the analysis results by developing a malware classification technique based on native-related features. This highlights that the proposed methodology can be helpful to both humans and machines, and the results are useful for categorizing surreptitious native code facets in modern malware samples. This chapter is based on the paper "The Dark Side of Native Code on Android" currently under submission at the ACM Transactions on Privacy and Security (TOPS) j.w.w. Alessio Merlo of the University of Genoa, and Andrea Possemato, Savino Dambra, Simone Aonzo, and Davide Balzarotti of EURECOM [184].

Chapter 5 presents the inotify-based state inference attack, its impact, and the countermeasures Android or a vendor should take to limit this issue. Moreover, it describes a practical use case, showing how an attacker can build a phishing attack without permission. This chapter is based on the paper "Android, Notify Me When It Is Time To Go Phishing" published at IEEE European Symposium on Security and Privacy (EuroS&P) 2023 j.w.w. Alessio Merlo of CASD – School for Advanced Defense Studies, and Andrea Possemato, Dario Nisi, and Simone Aonzo of EURECOM [185].

Chapter 6 describes *DroidDungeon*, a novel Android sandbox that jointly meets the anti-evasion, scalability, and maintainability requirements and enables us to perform the first analysis of evasive controls in modern Android applications. This chapter is based on the paper "Unmasking the Veiled: A Comprehensive Analysis of Android Evasive Malware" accepted to the ACM Asia Conference on Computer and Communications Security 2024

that will take place in July 2024 j.w.w. Alessio Merlo of CASD – School for Advanced Defense Studies, and Dario Nisi, Savino Dambra, Simone Aonzo, and Davide Balzarotti of EURECOM [186].

Finally, Chapter 7 concludes the summary and provides possible future research directions.

Chapter 2

Android: Background & Protection Techniques

This chapter introduces the background concepts needed to fully understand the technical contributions provided in the follow-up of the thesis. Since this thesis spans very different Android topics, ranging from Android app-level virtualization to anti-analysis techniques, this section focuses on notions common to every field, leaving a more detailed cover of the needed technical background to the individual chapters when necessary.

In the first part, this chapter introduces the primary block of the Android ecosystem: the application (hereafter, app). This section discusses the app's main components and the lifecycle from the developing phase to the installation on a user device. From the security standpoint, Android apps and the operating system are the two main objectives of malware authors: they mainly deliver the attacks through malicious apps that target the security posture of the entire device (e.g., gaining the superuser rights by, for instance, exploiting a vulnerability of the operating system) or the user by attacking other apps. Thus, it introduces and highlights different security aspects and challenges related to the Android platform and its apps.

In the second part, this chapter discusses the state-of-the-art mechanisms that an app – benign or malicious – could exploit to protect itself from static and dynamic analysis. It also proposes a comprehensive taxonomy of all evasive techniques. In detail, it provides a novel high-level per-purpose categorization of all documented anti-behavioral controls. Given that this part discusses only well-known techniques collected through blog posts, malware writeups, and scientific papers and does not aim at proposing novel protection techniques, it could be considered as part of the common background needed for the rest of the thesis.

2.1 Dissection of Android Apps

Android is an open-source, Linux-based software stack created for various devices (e.g., smartphones, tablets, TVs) and form factors. The Android ecosystem is much more complex than one may think: it comprises numerous layers and actors that cooperate to produce the final result, from kernel manufacturers to app developers. In general, all Android-based devices are built on top of the Android Open Source Project (AOSP), which is the free and open-source core of Android and provides the base features of the operating system. However, most devices run on the proprietary Android version developed by different vendors (e.g., Google, Samsung), which ship additional proprietary apps and customizations.

Apps are the cornerstone of the Android ecosystem and represent the primary vehicle for developers to provide new features to users. At the same time, apps are also the primary way malicious actors harm and attack the final Android users.

Regardless of the Android device, each app – system or third party ones – lives in its own *Application Sandbox*, a security mechanism to isolate every app from the others and protect them and the whole system from malicious actors. To do this, Android takes advantage of the underlying Linux kernel and user-based protection: Android assigns a unique user ID (UID) to each Android app and runs it in a separate process (each process has its virtual machine, so an app's code runs in isolation from other apps). Android sets permissions for all the files that belong to an app; thus, only the user ID assigned to it can access them. In this way, the kernel enforces the security between different apps and the system at the process level through standard Linux facilities, such as user and group IDs assigned to apps. By default, apps cannot interact with each other and have limited access to the OS. It is worth noticing that all software layers above the kernel (e.g., OS libraries, system, and third-party apps) run within the Application Sandbox.

2.1.1 Android Apps' Anatomy and Lifecycle

Anatomy of an app. An Android app is usually developed in Java or Kotlin[1] and leverages XML files to represent the graphical user interface (UI) or describe the metadata of the app. However, today, there are other ways to develop Android apps, for example, hybrid apps: they are mainly built on top of web technologies and run over a WebView. While very versatile, all these technologies are generally not optimal for performance-critical tasks or low-level

[1] From now on, we will explicitly refer to Java code, although the same considerations are valid also in the case of Kotlin-based apps.

interactions with the device hardware. To deal with this shortcoming, the Android Software Development Kit (SDK) also allows writing portions of an app through native (C/C++) code.

Although Android supports Java Virtual Machine-based (JVM) languages, Android apps' compilation process differs from regular Java apps. On Android, the Java code is first compiled into the corresponding Java bytecode, then compiled into the Dalvik-bytecode DEX (`.dex` extension). For the native component, instead, the Android system provides the Android Native Development Kit (NDK), a set of tools containing compilers, debuggers, and build systems that allow the developer to compile native code for their Android apps. At the end of the compilation process, the NDK generates native libraries as Executable and Linkable Format (ELF) files in the form of Shared Object (`.so` extension). Contrary to the DEX file format, which is architecture-independent, Android apps must ship a native library for each architecture they should support. The interaction between the Java bytecode and the native libraries, and vice versa, is made possible thanks to the Java Native Interface (JNI).

When all the code has been compiled, it is embedded in an Android app PacKage (APK), an archive containing all the necessary files to run the first execution of the app: DEX files and native libraries, images, and UI-related resources, and the *Manifest* file. The Manifest provides valuable information about the app, such as its unique package name (on Android, you cannot install two apps with the same name on a single device) and the required permissions.

Finally, each APK is signed with its developer's private key and contains the corresponding public certificate. This mechanism grants the APK file's integrity but cannot provide any authentication, as a trusted certificate authority does not need to issue the developer certificate.

Android permissions. Permissions allow Android to protect sensitive data (e.g., system state and a user's contact information) and prevent potentially dangerous actions (e.g., connecting to a Bluetooth device or recording audio). Android permissions are ranked into different protection levels [98] according to the sensitivity of the resource they protect. While the system automatically grants *normal* permissions (i.e., those that present minimal risk to the user's privacy) at installation time, *dangerous* (or *runtime*) permissions need explicit runtime authorization by the user. Each time an app tries to access a resource or perform an action protected by dangerous permissions, the user is prompted with a UI message detailing the type of resource/action the app attempted to access and asking whether to grant or deny it. Finally, the highest protection level is the privileged one. Permissions with this level have an impact on the overall security posture of the device. Thus, the user has to manually grant

them through a multistep procedure in the System Settings app, during which the system warns her about the risks of this permission.

App distribution. Android apps are mainly delivered through proper software repositories, known as app stores, which allow developers to publish their APK files. While iOS relies on a single and centralized market (i.e., the Apple App Store), which Apple directly controls, Android allows the distribution of the same app on dozens of different app stores beyond the official one (i.e., the Google Play Store). The most popular app stores differ from region to region. For instance, in Europe and the US, the Google Play Store has the most significant market share, while in China, its market share is less than 4%, and MyApp (Tencent) currently holds 25% of the market. Moreover, Android allows the distribution and installation of APK files directly (e.g., through email or URLs) without uploading the apps to an app store.

App installation. Whenever a user wants to install an app, Android performs a soft check on the app certificate before installation (recall that the developer must sign an APK with her certificate). It is worth noticing that Android continues the installation even if the signer of the developer certificate is unknown or it is self-signed. Therefore, Android performs only primary verification on the APK structure and its integrity, thereby leaving the decision to install the app (and, consequently, to trust the source of the app) to the final user.

Then, at installation time, Android assigns a new user and group ID to the app and creates a dedicated directory for it in `/data/app/` folder in which the system copies the original APK, renaming it as "base.apk". This directory's name – also known as *installation path* – consists of the app package name and a random string since Android 8.0 [94]. The random component prevents a malicious app from trivially determining another app installation path and accessing its private files. Thus, according to the Android documentation, the only supported way to obtain a third-party app installation path is through the `getPackageInfo` API of the Package Manager, which is subject to the permission mechanism.

For example, suppose to install an app whose package name is *com.example* on a device with an x86 CPU. A legit installation path could be `/data/app/com.example-Bd6GIb47XTzpL16==/`. In this location, the system copies the app's APK (renaming it `base.apk` file) and creates a subdirectory named `lib/x86/` that contains all the native libraries (SO files) for the x86 architecture.

Finally, when the APK is installed on an Android device, another compilation step – that only affects DEX files – takes place on the device: dalvik-bytecode files are Ahead-Of-Time (AOT) compiled to generate an executable app for the target device architecture. In particular, since Android 5.0, Google introduced the Android RunTime (ART) to replace the Dalvik Virtual Machine (DVM). While Dalvik just-in-time (JIT) compiles framework code and

apps on demand, ART uses a hybrid approach that combines AOT, JIT, and profile-guided compilations. After the first execution and when a device charges, ART performs the AOT compilation of all frequently used code based on a profile generated during the first runs. Thus, during the next executions, ART uses the profile-guided code and avoids doing JIT compilation at runtime for methods already compiled. Methods that get JIT-compiled during the new runs are added to the profile, which the following compilation will pick up. It is worth noting that ART performs AOT compilation also for the framework libraries. However, in this case, the amount of compiled code depends on configuration options specified when the framework is built [99].

This approach brings numerous improvements in terms of performance and battery life at the price of having longer installation times and more extensive storage requirements: since the bytecode has been compiled, the app will not require extensive CPU usage for just-in-time optimizations.

The AOT compiled code is saved in a particular file format named OAT, a custom ELF executable that includes two special sections: `oatdata` to store headers and info about the compiled DEX files, and `oatexec` to store the compiled code. It is worth noticing that the OAT file format changes between Android versions, and no official documentation tracks those changes.

Native libraries, on the other hand, are not affected by this additional optimization step; in fact, they are already compiled for the architecture(s) in which the app will run. This means that if an app wants to be installed on several devices that differ in Application Binary Interface (ABI) and Instruction Set Architecture (ISA), it must contain native components compiled for each target architecture it wants to support. To date, Android supports the following ABIs: `armeabi-v7a, arm64-v8a, x86, x86_64`; however in the past, it supported also `ARMv5 (armeabi)`, and `32-bit` and `64-bit MIPS`, but they are no longer supported [95].

Zygote & app startup. Zygote is the parent process of all Android apps, and the init process creates it during the system boot. This process initializes the first instance of DVM and pre-loads all framework classes the apps should use very often. Each new app process is a fork of the Zygote process, which is used as a template, thus saving the time required to load these resources into its address space. In this way, the memory address space of the Android framework and the native libraries are the same for all Android apps (inherited from the Zygote).

2.2 Anti-analysis techniques

An evasive control is a technique used by an app to prevent the runtime inspection and analysis of its behavior. The MITRE Malware Behavior Catalog [46] classifies this as a malware objective under the name of *Anti-Behavioral Analysis*. It is essential to distinguish it from other forms of code protection (e.g., code obfuscation and Dynamic Code Loading) that are instead classified as *Anti-Static Analysis*[2]. While the anti-behavioral techniques aim at concealing the action performed by a sample, anti-static analysis techniques focus on protecting the app code by making it more complex to analyze. As such, they do not affect the app's runtime behavior, which continues to execute similarly in any environment. Therefore, the two objectives – evasive controls and static analysis protections – are orthogonal and often combined together [175]. For instance, Denuvo Mobile Game Protection jointly uses evasive checks (e.g., anti-debugging to verify the presence of a debugger) and dynamic code loading to protect Android apps from repackaging [116].

2.2.1 Anti-static analysis

Anti-static analysis refers to behaviors and code modifications that prevent or hinder static analysis of a sample. Briefly, static analysis directly analyzes the compiled file (i.e., DEX or ELF files) or APK's features, such as embedded strings, header information, or the apps' code's control flow and call graphs. Thus, anti-static analysis techniques aim to complicate an app's manual and automatic reverse engineering process without introducing any anti-tampering check: they modify code portions or remove hard-coded strings and identifiers.

Code obfuscation. Obfuscation approaches modify an app's bytecode or source code without changing its behavior to make some manual or automatic analysis more difficult. There exist a lot of off-the-shelf obfuscation tools, both open source (e.g., [17]) and commercial (e.g., [110]). Obfuscation comprises many techniques that have different objectives in protecting the APK. For instance, code or resource encryption aims to hide some APK's metadata or resource, such as hardcoded string, by replacing them with their encrypted value (decrypted only during the app execution). Conversely, the reflection technique hinders the creation of a call graph by static analysis tools, replacing direct method invocations with a reflective version. It is worth noticing that a user can combine different obfuscation techniques (e.g., reflection and encryption) to ensure greater robustness against static analysis.

[2]The Malware Behavioral Catalog currently focuses mainly on Windows malware and does not contain specific checks for Android. However, it is helpful to adopt its naming scheme to present the work better.

Code virtualization. Code virtualization is a specific obfuscation scheme in which parts of the original executable code are mapped to and transformed in semantically equivalent virtual instructions. At execution time, the bytecode is emulated by an embedded virtual machine on the actual device. The new instructions can be designed independently; thus, the bytecode and interpreter significantly differ from those in every protected instance. In this way, the program's original code never reappears.

Two different approaches are proposed in [112] and [250]; the first works at the DEX level, while the latter focuses on native libraries.

Dynamic Code Loading & Modification. DCL is the practice of allowing an app to load additional code at runtime and execute it. Such behavior allows bypassing the 64K reference limit [53] imposed by the DEX file format[3]. Also, the loaded code could be remotely fetched or locally unpacked, and it is not part of the APK's initial codebase; thus, static analysis can not consider this code as it is not static available. Despite several benefits (e.g., high code reuse and performance improvements as the code is only loaded when necessary), from a security standpoint, dynamic code loading drastically complicates the analysis of Android apps. Therefore, the combined use of DCL and evasion techniques allows malware samples to dynamically load malicious code only when they are sure they are not under analysis.

On the other hand, dynamic code modification refers to the self-modifying code practice in which the app alters its instructions during the execution. Thus, dynamic code is generated (or modified) at runtime instead of being fixed at compile-time, bypassing static analysis techniques.

2.2.2 Anti-Behavioral Analysis

Over the years, researchers studied different aspects related to evasive controls, such as their adoption in benign samples [210, 29, 149, 48] or their impact on malware classification [168, 147, 57, 212, 151, 27, 144, 4, 47, 26]. Moreover, novel and more sophisticated anti-behavioral techniques have been routinely presented year after year [227, 187, 119, 198, 249, 128, 196, 200, 60, 189, 73, 58, 33, 152, 109, 52].

Thus, at the beginning of this work, we collected all the documented evasive techniques by searching through blog posts, malware writeups, and scientific papers.

A first attempt to propose a taxonomy of protection techniques used in Android apps was recently published in 2023 by Faruki et al. [63]. However, the authors considered only

[3]The Android app bytecode can reference up to 65,536 methods, among which the Android framework methods, the library methods, and the app's methods

a small subset of the techniques outlined in this section, focusing mainly on obfuscation. As already explained, this thesis follows the MITRE classification instead and, therefore, does not consider obfuscation as part of the anti-behavioral controls but includes it in the anti-analysis techniques. Thus, to better study evasive techniques, this thesis proposes a novel categorization based on the purposes of the controls. The rest of this section presents the list of evasive behaviors covered in this study by grouping them into three high-level macro-categories: *Environment verification*, *APK tampering verification*, and *High-level verification*.

Environment verification

Environment checks aim to detect the reliability of the environment in which apps are installed.

Root detection. The Android design does not require users to use the root account; therefore, such an account is disabled by default. A method known as *rooting* allows an end user to get super-user access to an Android smartphone. Super-users can alter system settings, access private areas in the primary memory, install specialized apps, or use a debugger or dynamic analysis tool. Therefore, the presence of executables that require root permissions may indicate that a sample is executed in an instrumented environment, such as a sandbox. Of all the possible tests that can be used for root detection, the most common look for the presence in the file system of the `su` or `busybox` executables or test whether well-known paths that are usually read-only have write permission [191, 196, 210].

Debugging detection. A debugger introduces changes to the memory space of the target process and may impact the execution time of certain code snippets [29, 149]. Thus, anti-debugging techniques can detect (by looking for specific artifacts or side effects) or prevent the app from being debugged.

Hook detection. Anti-hooking controls aim to detect dynamic binary instrumentation tools (e.g., Xposed [111] and Frida [157]) that can hook and tamper with the execution flow of an app. The simplest way to detect their presence is by scanning package names, files, or binaries and looking for well-known frameworks' resources. In addition, each dynamic analysis framework works differently and may require specific detection techniques. For instance, Xposed is an Android app that applies modules directly to the Android OS ROM and requires root privileges. Contrary, Frida injects instead a JavaScript engine into the instrumented process.

Emulator detection. Anti-emulation techniques try to detect whether the app is not running on an actual device. For instance, the Android emulator is built on top of the QEMU [40] emulator, and an emulator may not provide the same hardware functionalities as an actual phone. (for instance, for sensors like gyroscope and accelerometer), or some particular artifacts may or may not be present (e.g., different files or file content, Android system properties). In addition, some emulators do not fully support the Google Play Services (e.g., Genymotion [77]) or require some changes in the system property (e.g., `ro.build.tag`).

Memory integrity verification. This type of evasive control aims to verify the integrity of the app's memory space against memory patches applied at runtime [196]. For instance, hooks to C/C++ code can be installed by overwriting function pointers in memory or patching parts of the function code (e.g., inline hooks that modify the function prologue). Thus, an app can check the integrity of its memory regions to detect any alteration.

App-level virtualization detection. Android virtualization is a recent technique that enables an app (*container*) to create a virtual environment in which other apps (*plugins*) can run while fully preserving their functionalities [214, 139]. The container app acts like a proxy, intercepting each request from the plugin app to the Android operating system and vice versa to fool the OS into believing that the container issued the request. Anti-virtualization techniques aim to detect whether the app is executed within these virtual environments [247, 198, 141, 239, 50]. This could be done in different ways, for instance, by verifying its own UID, the number of running processes, or the object instance of the Android API clients.

Network artifact detection. This type of evasive control aims to inspect the network interfaces to detect artifacts, such as unusual interface names or ADB connected over the network. Moreover, since sandboxes often intercept and analyze the app's network traffic to understand its behavior, evasive apps may also check for the presence of VPNs or proxies.

APK tampering verification

Anti-tampering (AT) techniques detect any modification on the original app during its execution [29, 149]. If modifications are detected, the app can take evasive actions, such as turning off certain features or terminating its execution.

Signature checking. Given that APKs are digitally signed, this control checks if the certificate is the expected one.

Code integrity. These checks verify whether some code or resource has been tampered with by computing its signature at runtime and comparing it with pre-computed and hardcoded values.

Installer verification. Since API level 5, the Android Package Manager stores information on which 'installer' app (e.g., Google Play Store or Samsung Store) was used to install any given app. These evasive check retrieves the package name of the installer app to verify whether the app has been installed from the expected app store. Tampered apps are more likely to be distributed on unofficial app stores that differ from the original ones. Moreover, an APK can also be downloaded directly from a website, and thus, in this case, the installer app can be a browser or a file manager.

High-level verification

SafetyNet attestation & Integrity API. SafetyNet [104] is a platform security service offered by Google that provides a set of APIs to help protect apps against security threats, such as device tampering and other potentially harmful apps. For instance, to verify the integrity of a device, an app leverages the Attestation API by invoking the `attest` method of the SafetyNet client, while to check if malicious apps are installed on the device, an app can invoke the `listHarmfulApps` API.

Starting January 2023, the SafetyNet attestation is deprecated and replaced by the Play Integrity API [102]. It offers an enhanced security mechanism that verifies the app's integrity to defend against tampering and redistribution of your app and the environment in which it is running. Also, it consolidates multiple integrity offerings (including the ones offered by SafetyNet) under a single API.

Human Interaction. Even sophisticated Android malware sandboxes often neglect to mimic realistic user behavior and interaction. In 2022, Kondracki et al. [128] have shown how user-related artifacts (e.g., number of photos and songs, list of contacts) can be abused to distinguish an actual device from a sandbox environment.

Chapter 3

Repack Me If You Can: An Anti-Repackaging Solution Based on Android Virtualization

3.1 Introduction

The Open Worldwide Application Security Project (OWASP) is a non-profit foundation for improving software security. It operates under an open community model, meaning anyone can participate in and contribute to OWASP-related online chats and projects. The OWASP Top Ten, first published in 2003, aims to raise awareness about app security by identifying some of the most critical risks facing organizations, and it is regularly updated every about three or four years. Three of the top ten mobile risks in 2016 are related to man-at-the-end issues, such as reverse engineering and code tampering [165]. Those attack vectors are merged in a single threat named 'Insufficient Binary Protections' in the top ten list in 2023. In this scenario, besides collecting security-relevant information (e.g., commercial API keys or hardcoded cryptographic secrets), attackers could also manipulate app binaries to access paid features for free or to bypass other security checks; thus, attackers carry out what is nowadays known as repackaging attacks.

Traditional repackaging attacks aim to lure the user into installing a malicious app that looks legitimate. In particular, such attacks revolve around the following steps: reverse engineering the target app, injecting malicious code, recompiling and distributing the modified app on the app markets or through other channels. Given the similarity between the repackaged and the original apps, the user cannot distinguish between them and can be fooled

into installing the malicious one. Once the malicious app runs on the victim's phone, it can start executing the attacker's code.

A recent growing trend concerns the exploitation of the Android virtualization technique [50] to generate and distribute malicious repackaged samples more efficiently than traditional repackaging attacks. Android virtualization enables an app (i.e., *container*) to create a virtual environment, separated from the Android default one, in which other apps (i.e., *plugins*) can run while fully preserving their functionalities.

The container has complete control over its plugins. In detail, it can execute any app in its virtual environment (i.e., even apps not installed on the actual device) and control its runtime behavior and interaction with the Android framework API. Thus, a maliciously crafted container could inject arbitrary code inside the running plugins and modify - or even block - their API calls, passing unnoticed to the apps.

The popularity of the virtualization technique, confirmed by the number of downloads of the virtualization apps on the Google Play Store [217, 115, 216], is given by its primary use case scenario, i.e., running multiple instances of the same app on a single device. For instance, if a user has two separate Telegram accounts and wants to use them simultaneously, she can run the first in the Android environment and the second in one of the virtualized contexts. Besides legitimate uses, the virtualization technique has already paved the way for threatening attacks [249, 247, 198, 50].

Concerning traditional repackaging attacks, virtualization-based ones are easier to setup. The former requires the attacker to I) decompile the APK of the victim app, II) detect and, in case, remove anti-repackaging protections, III) inject some malicious code, and IV) recompile it. Instead, virtualization-based repackaging attacks allow the attacker to make the container execute the original victim app and some malicious code without modifying the original APK. The malicious code can be part of the container logic or another plugin running in the same virtual environment as the victim app. Two examples of virtualization-based malware have already been detected in the wild [23, 25]. In the first one, the malware targets the Twitter app and aims to support the app's dual-instance execution. The user logs into his second Twitter account inside the dual-instance app, which can also hijack user input besides creating a virtual environment. The second malware aims to distribute an updated version of the WhatsApp app, similar to the original one, besides the malicious code crafted to steal users' sensitive data. The virtualization technique is confirmed by the presence of a file called `WhatsApp.apk` inside the malicious container APK, which is loaded at runtime as a plugin.

To defend against malicious usage of the virtualization technique, researchers have put forward some anti-behavioral solutions [239, 198, 50, 247, 141, 199], known as anti-

virtualization techniques (refer to Section 2.2). Most of them are distributed as a library that plugins must embed to detect whether they run in a virtual environment at runtime. Limitations of such approaches are twofold: the checks they rely on can be easily bypassed [9], and they cannot distinguish between benign and malicious usage of the virtualization technique. On the contrary, only two solutions [247, 199] have been proposed to defend against virtualization-based repackaging attacks specifically. Both of them rely on a static analysis approach to find, first, any usage of the virtualization technique and, then, the purpose of its use. Being designed as static analysis tools, they are affected by well-known limitations (e.g., missing evaluation of code dynamically loaded at runtime).

Given the number of malware samples designed on top of the virtualization technique, there is an urgent need to provide a reliable defense methodology that can I) prevent any repackaging attack (both traditional and virtualization-based) and II) be adopted by any Android user straightforwardly. To this aim, this chapter presents *MARVEL* (i.e., Mobile-app Anti-Repackaging for Virtual Environments Locking), the first dynamic anti-repackaging solution based on the virtualization technique. *MARVEL* relies on a *trusted container* that creates a virtual environment where plugins equipped with proper anti-repackaging controls run. Plugins cannot run in the native Android OS due to our anti-repackaging checks based on APK tampering verification, environment evaluation, and anti-debugging techniques. Thus, we extended such controls to fail if they are not executed on the trusted container, which is in charge of unlocking the anti-repackaging checks. It is worth noticing that *MARVEL* combines anti-static and anti-behavioral techniques to hide the anti-repackaging controls.

To experimentally evaluate the feasibility of *MARVEL*, it has been implemented in a tool (i.e., *MARVELoid*) to test the methodology against a dataset of $4,000$ apps. In particular, each app is protected with 24 different configurations of the protection parameters, ending up with 96k different protected samples. The tool achieved a 97.3% success rate and required - on average - only 98 seconds per app to introduce the protections. Moreover, *MARVELoid* evaluated the protected apps at runtime, measuring the number of failures in their execution and the overhead introduced by *MARVEL* regarding CPU and required memory. The results showed that *MARVEL* introduces a limited overhead concerning traditional virtualization techniques, i.e., an increment - on average - up to 4.7 percentage points (pp) in the CPU usage and 0.2 pp for the consumed memory.

In summary, this study has advanced research in the area of application security through the following contributions:

- *MARVEL*, the first anti-repackaging solution that dynamically prevents any repackaging attack and protects Android users by running on mobile devices;

- *MARVELoid*, a tool to enforce the *MARVEL* protection and experimentally evaluate its effectiveness;
- a thorough empirical evaluation of the proposed methodology over $4,000$ apps, achieving a 97.3% success rate in protecting the apps and introducing an average increment up to 4.7 percentage points (pp) in the CPU usage and 0.2 pp in consumed memory concerning existing virtual environments.

3.2 Background

This section introduces the threat model of the traditional repackaging attack, the state-of-the-art protection mechanisms, and the internal details of an Android virtual environment.

3.2.1 Android App Repackaging

A repackaging attack aims to distribute a modified version of an app that contains a malicious payload to, for instance, unlock some premium features or distribute malicious code. The victim is a general Android user, enjoying her mobile device and regularly downloading apps from any app store (e.g., Google Play Store and Samsung App Store).

The threat model for traditional app repackaging involves three actors: the developer, the attacker, and the final user. In a utopian (i.e., attacker-free) scenario, the developer implements and delivers the app to the users through one or more app stores. Thus, the user fetches the app and downloads it. At installation time, Android verifies the signature and the integrity of the APK file and installs it on the user's device without ensuring any authentication mechanism.

Nonetheless, in a real scenario, the developer applies some anti-repackaging protection schemes during the app development to deter the attacker who tries to repackage the app. Anti-repackaging techniques may inject APK tampering verification controls that ensure the legitimacy of the APK file and other evasive controls to make the runtime analysis of the app more challenging. Thus, to carry out the traditional repackaging attack, the attacker must encompass the steps illustrated in Figure 3.1.

From her side, the attacker accesses the APK file directly from the app store, behaving like any regular user: she downloads the app from an app store, analyzes and reverses the APK through static and dynamic techniques (Step 1), and tries to detect and turn off potentials anti-repackaging controls (Step 2). If at least a match is found, the attacker needs to try deactivating the protections (Step 3); otherwise, she can directly move to the next step. Then,

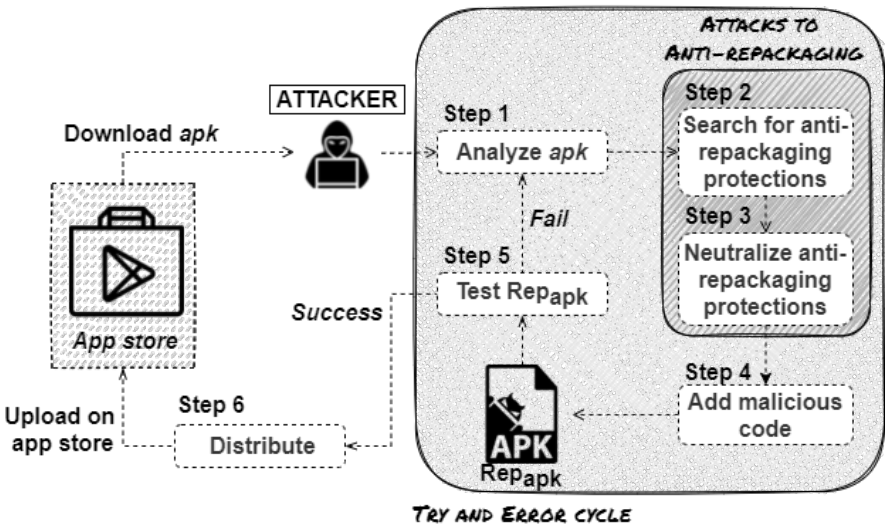


Figure 3.1 Threat model for the standard app repackaging attack.

once all the detected anti-repackaging techniques are disabled (if any), the attacker adds some malicious payload to the app or modifies its content (Step 4) and builds and signs the Rep_{apk} APK file, a repackaged version of the original app (Step 5). Finally, the attacker tests whether Rep_{apk} works properly: if this is the case, the attacker redistributes the repackaged app both on app stores and through side channels (e.g., by email, web servers or leveraging phishing attacks), aiming to force the user to install it instead of the original app (Step 6). Otherwise, the attacker must conduct further analysis and repeat Steps 1 to 5 (the "try and error cycle") to detect and turn off other anti-repackaging protections on the original APK.

Defence mechanisms against repackaging. In recent years, researchers proposed several techniques [133, 149] to discourage attackers from repackaging apps. Such techniques can be divided into two main categories based on their purposes, namely *repackaging detection* and *anti-repackaging* (also known as *repackaging avoidance* or *self-protection*).

The first group of techniques aims to recognize apps that have already been successfully repackaged and delivered to app stores or users' devices. Common repackaging detection solutions rely on app similarity, which is evaluated through a two-step process: each app is first profiled according to a set of distinctive features (e.g., call graph and control flow graph [142]), and then, apps feature profiles are compared to each other and, if their similarity is higher than a given optimal threshold (calculated according to some machine learning techniques, either supervised [222] or unsupervised [80]), the apps are considered clones, thereby indicating that one is probably the repackaged version of the other one.

The latter group (i.e., *anti-repackaging*) comprises techniques to make the repackaging process more difficult for an attacker. In particular, the app developer injects *detection nodes* that implement APK tampering controls to verify the authenticity and the integrity of the sample, for instance, by leveraging tampering verification controls and checking the metadata of an app (e.g., the APK signature or its package name). If the anti-tampering controls fail at runtime, the AT mechanism commonly throws a security exception to force the app to crash [29]. Thus, an attacker has to find and turn off all detection nodes to repackage the app successfully.

In addition, anti-repackaging solutions can further enhance protection by preventing the reverse-engineering of the detection nodes by combining evasive techniques (e.g., anti-debugging) with anti-static analysis mechanisms (e.g., obfuscation). Among those, some approaches rely on *logic bombs*, practices regularly used in malware. The logic bomb pattern proposed by [246] protects (i.e., hides) AT controls by encrypting their code using constant values contained in the program as encryption keys, exploiting the differences between an attacker and the developer's knowledge. Moreover, the encryption keys are removed from the code and replaced with the corresponding pre-computed hash values. Thus, the protection of the logic bomb is granted by the one-way property of cryptographic hash functions, which makes it hard for the attacker to retrieve the original value (i.e., the decryption key) from its hash. To this aim, an attacker must leverage both static and dynamic analysis techniques to execute all bombs, intercept the decryption keys, get access to the anti-tampering controls, and remove or deactivate them.

Overall, an ideal anti-repackaging scheme never allows the attacker to obtain a working repackaged app; however, actually, an anti-repackaging technique is considered reliable if the complexity for the attacker to remove all protections in an app is more time-consuming than developing the same app from scratch. As a final observation, repackaging detection techniques are instead applied on the app store and, sometimes, on the mobile device to detect already repackaged apps.

3.2.2 Android App-level Virtualization

The Android virtualization technique enables an app – known as *container* – to create a virtual environment where other apps – *plugins* – can run. A user can install several containers, which generate their corresponding virtual environment, in which the user can execute several plugins independently from the underlying Android OS and other virtual environments. DroidPlugin [214] and VirtualApp [139] are the two most well-known frameworks supporting

the generation of Android virtual environments and share a common design. A virtual environment can run any Android app, single or multiple apps at a time, and apps not installed on the device. Furthermore, it does not require any additional privileges enabled on the device (e.g., root privileges).

The virtualization technique relies on Dynamic Code Loading and Java dynamic proxy [161]. The former enables an Android app to load external code from a DEX, JAR, or even APK file of another app. On the other hand, the Java dynamic proxy allows the creation of a wrapper object that intercepts all method calls toward a specific object instance, eventually adding some functionality.

Figure 3.2 shows the internal architecture of an app generating a virtual environment for a plugin. The architecture involves the following components: the Container App (i.e., container), the Plugin APK (i.e., plugin), the Dynamic Proxy Module, and the ClassLoader.

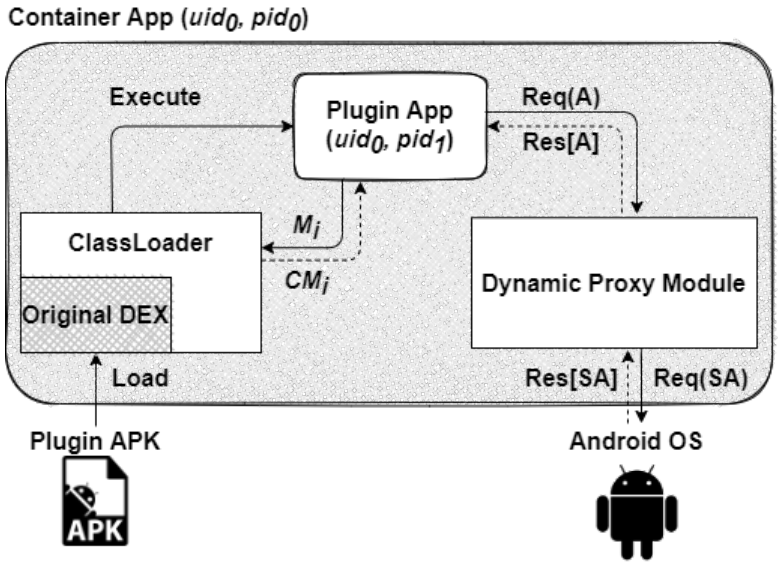


Figure 3.2 The internal structure of an Android virtual environment.

First, the container extracts the bytecode from the APK of the plugin app (i.e., the `classes.dex` file(s)) and loads them into the ClassLoader of the container. Then, it forks its process (i.e., pid_0) into a new process to host the plugin (i.e., pid_1). It is worth noticing that the container and the plugin share the same UID (i.e., uid_0); thus, they share the same security policies and permissions. At runtime, the ClassLoader resolves classes and methods (i.e., M_i) for the plugin by returning the appropriate code (CM_i). Moreover, the container has to manage the lifecycle of all the plugin components. In detail, Android apps contain several components (i.e., Activities, Services, Broadcast Receivers, and Content Providers), which must be declared in the Manifest file so the Android OS can register them at the installation

time. The management of each component involves an interaction between the app and the Android operating system: for example, to show an Activity (*A*) on the screen of the smartphone, an app has to send a request to open that Activity (`startActivity(`*A*`)`) to the Android OS, which, in turn, has to receive a reply containing the details of the device where the app is running (e.g., the dimension of the screen). When the plugins run in a virtual environment, the Android OS receives all requests from the container. Thus, the container must declare the same components as the plugins running in its virtual environment. To this aim, the container declares a generic number of *stub* components. Then, it can rely on the Dynamic Proxy Module to intercept each request (or reply) going towards (or coming from) the Android OS to dynamically change the component's name. In the example of Figure 3.2, the Proxy module creates a Stub Activity (*SA*) and sends the modified call to the Android OS (`startActivity(`*SA*`)`). Then, the result of the invocation (*Res*[*SA*]) is mapped back to the original component (*Res*[*A*]). Concerning permissions, the container requires all existing Android permissions to handle generic plugin apps.

3.3 Related Work

This section briefly summarizes the related works on anti-repackaging based on internal anti-tampering controls, and then it discusses details on anti-virtualization behaviors. Finally, it discusses some limitations of the proposed techniques concerning virtualization-based repackaging attacks.

Anti-Repackaging. The first anti-repackaging technique for mobile apps was proposed in 2015 by Protsenko et al. in [173]. The main idea is to encrypt the `classes.dex` files in the APK and dynamically decrypt them during the app execution. In 2016, Luo et al. in [140] proposed *SSN*, which injects a set of detection nodes into the app code (i.e., through the Java bytecode). The detection nodes hide a stochastic function that aims to detect tampering and, in case, force the app to crash. In 2017, Song et al. [201] proposed an app-reinforcing framework named AppIs. The main idea is to create a graph of security units (i.e., detection nodes), which performs I) AT controls to detect repackaging and II) integrity checks of other security units. In 2018, Chen et al. [38] proposed a *SDC* scheme, encrypting pieces of code. The ciphered portion of code is decrypted and executed at runtime only if the app is not repackaged. Each piece of code is encrypted with a different key, which is related to the anti-tampering controls that are activated at runtime. In the same year, Zeng et al. [246] designed BombDroid, a defending technique that leverages logic bomb as an anti-repackaging protection for Android apps. Similar to SDC, BombDroid hides different AT controls inside

the encrypted code of a logic bomb. In 2019, Tanner et al. [213] proposed an extension of BombDroid, which aims to implement the logic bombs in the native code. In 2021, Merlo et al. [149] defined some guidelines for reliable anti-repackaging techniques and proposed a new anti-repackaging methodology, i.e., ARMAND [148] that leverages a pseudo-stochastic criterion to inject multiple types of logic bombs both in the Java bytecode and in the native code.

Anti-Virtualization. Malicious exploitation of the virtualization technique [249, 247, 198, 50] has motivated researchers towards the design of possible defense mechanisms in the form of anti-behavioral analysis techniques [239, 198, 50, 247, 141, 199], which, to some extent, might also defend from virtualization-based repackaging attacks. Most are designed to detect whether an app runs in a virtual environment. To achieve this aim, such solutions are included as a library that evaluates specific features of the protected app at runtime.

A virtual environment provides the plugin with a different context concerning the native Android OS that affects the following elements: permissions, number and name of the components, processes name, organization of the internal storage of a plugin, and private data sharing among plugins. For example, a plugin can be granted more permissions than the declared ones since sharing the UID between the container and the plugin also involves sharing permissions. Another example refers to the number and names of the components the container declares, which uses stub components to wrap the plugin ones. This design implies that the Android OS is only aware of the existence of the container components (i.e., the components of the plugins go unnoticed). All existing defense mechanisms against the virtualization technique check the features mentioned above by either evaluating the return value of specific Android APIs or by inspecting the OS itself (e.g., the `proc` directory) and force the plugin to crash if the return values or the Android OS features are not the expected ones.

So far, only two solutions [247, 199] have been specifically designed to detect virtualization-based repackaging attacks. Both share the same purpose: first, identifying whether an app uses the virtualization technique and, then, whether its usage is for malicious purposes. Although specific detection mechanisms differ, both solutions rely on a static analysis approach. The authors of [247] identify a virtual environment according to the similarity among the declared components (containers usually declare components - named *stub* components - having a very repetitive structure). On the contrary, [199] relies on control-flow graphs to see whether a plugin component is eventually replaced by a stub one. Both approaches also differ in the way they recognize a malicious use of virtualization: while the authors of [247] search

for a mismatch in the signature of both the plugin and the container, [199] check whether there is any stealthy loading of code.

3.3.1 SOTA vs. Android app-level Virtualization

First, all the proposed anti-repackaging schemes allow for protecting apps from traditional repackaging to different extents. Still, they could be more effective against repackaging through virtualization-based techniques (as discussed in Section 3.4).

Second, the anti-virtualization solutions discussed above are ineffective against a malicious container. Due to its proxy role, the container can intercept any call from the plugin (thus, from the library executing within the plugin) towards the Android OS and tamper with the response, sending back the expected value instead of the original one. Consequently, the same solutions do not protect from virtualization-based repackaging attacks.

Moreover, even though addressing virtualization-based repackaging attacks, the only two solutions designed for this attack (i.e., [247, 199]) share common limitations. First, being static analysis tools, they cannot evaluate dynamically loaded code at runtime, although they can detect the usage of the dynamic loading technique. In addition, their usage requires a non-negligible manual effort for the setup and configuration, thus limiting their adoption in the wild. Finally, state-of-the-art approaches focus on detecting already repackaged apps, i.e., the ones equipped with anti-repackaging mechanisms that attackers have successfully bypassed. On the contrary, *MARVEL* aims to prevent repackaging attempts by protecting the apps.

3.4 Assumptions and Requirements

This section presents the threat model for repackaging attacks through app-level virtualization and highlights the differences concerning the original ones. Then, it defines the security requirements for a reliable anti-repackaging scheme to meet w.r.t. a general – traditional or virtualization-based – repackaging attacks.

3.4.1 Virtualization-based Repackaging Attacks

Along with the traditional attack version, nowadays, an attacker can exploit the app-level virtualization to carry out a new repackaging attack. To achieve her purpose, we assume the attacker can own a mobile device, eventually a rooted/customized one, and can rely on

static and dynamic analysis techniques, as well as network traffic analysis ones, to inspect the behavior of the victim app.

A virtualization-based repackaging attack is depicted in Figure 3.3. This attack is far less complex than the traditional one and much more powerful. A container can execute any external app as a plugin without modifying its APK file, thus passing undetected to traditional anti-repackaging schemes and anti-tampering controls. Furthermore, its proxy capabilities allow intercepting all the API invocations to the Android OS and tampering with the responses (or the requests), thus overcoming the existing anti-virtualization solutions [9].

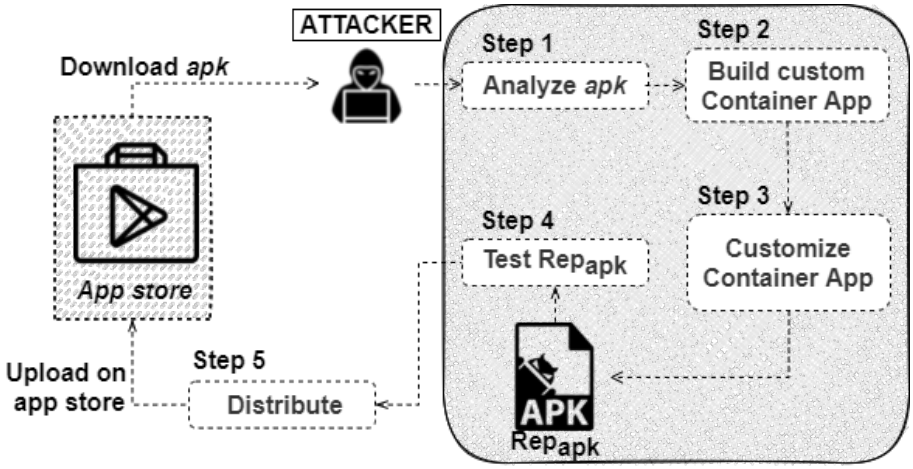


Figure 3.3 Threat model for virtualization-based repackaging attack.

To carry out a virtualization-based repackaging attack, the attacker analyzes the APK file of the victim app (Step 1) and builds a container to run it (Step 2). Then, she customizes the container to meet her purposes, such as stealing sensitive data from the victim app or unlocking some premium features (Step 3). Finally, the attacker tests the execution of the victim app in the customized container (Step 4) and distributes it by uploading the malicious APK to an app store (Step 5).

3.4.2 Security Requirements

The inner design of the virtualization-based repackaging attacks allows bypassing all the existing defense solutions: similarity checks fail since the malicious container dynamically loads the victim app at runtime, thus bypassing any static analysis. In the same way, anti-repackaging solutions fail since the malicious container I) does not modify the code of the victim app, thus neglecting tampering checks (e.g., signature verification), and II) can

intercept and tamper with any interaction with the Android operating system, bypassing dynamic environment controls (e.g., anti-virtualization techniques).

To overcome previous limitations, we advocate that a solution able to prevent and detect both traditional and virtualization-based repackaging attacks reliably should meet the following requirements:

(*R*1) Preventing the attacker from being able to statically analyze an app to fully reconstruct both the code and the protection controls.

(*R*2) Preventing an app from being executed in a malicious container that bypasses traditional AT checks.

(*R*3) Detecting an intermediate malicious container run in the container's virtual environment and executing a plugin.

3.5 The *MARVEL* Protection Scheme

MARVEL is an anti-repackaging protection scheme that leverages the virtualization technique to prevent both traditional and virtualization-based repackaging attacks. Furthermore, *MARVEL* fulfills the security requirements discussed in Section 3.4.

First, this section provides an overview of the *MARVEL* anti-repackaging scheme and, then, details its building blocks, namely code splitting and the Interconnected Anti-Tampering (IAT) controls. Finally, it discusses how *MARVEL* works at runtime.

3.5.1 Overview

MARVEL encompasses a mobile device running any recent version of the Android OS (i.e., Android API level $>= 26$) and a container. Furthermore, *MARVEL* requires that the protected apps can be successfully executed only as plugins in a Trusted virtual Container (TC), which is responsible for unlocking the anti-repackaging protections. In our scenario, we suppose that the container is a service that does not require root privileges, like the Google Play Services (GPS) [101]: during the first execution, an app verifies if the trusted container is installed (e.g., apps use the `isGooglePlayServicesAvailable` method to verify the availability of GPS) and, if this is the case, it requires to be executed by the trusted container. Mobile vendors already adopted a similar approach; for instance, Samsung Knox [190] is a commercial solution for Mobile Device Management that exploits a pre-installed trusted app that manages corporate data and apps.

MARVEL protects plugins through code splitting and interconnected anti-tampering controls. The former allows removing portions of code from the original plugin app, thus introducing mitigation against static analysis inspection (i.e., requirement *R*1). The latter involves improving anti-behavioral controls, evaluated during the interaction between the container and the plugin. Thus, the controls fail if the plugin is not executed in the expected container (i.e., requirement *R*2) or the container cannot communicate with the intended plugin (i.e., requirement *R*3). Furthermore, *MARVEL* enables the use of combined protection patterns, e.g., the injection of an IAT in the code of a method extracted using code splitting. Such a composition increases the difficulty of bypassing the protections, as the attacker is forced to execute and reconstruct the entire package before trying to locate and deactivate all the IAT controls.

3.5.2 Code Splitting

The code splitting techniques allow the partitioning of computations of programs on different nodes to relieve resource-constrained nodes from heavy calculations [177] or to avoid the disclosure of sensitive code/results [10]. In particular, code splitting suits a client-server environment, where the remote server is usually the preferred node for sensitive computations. At the same time, only authenticated clients are allowed to access the results [36, 72].

MARVEL applies code splitting between the trusted container and the plugin(s) to fulfill the *R1* requirement. Figure 3.4 shows the transformation process of a piece of code to which the code splitting technique is applied. *MARVEL* selects some methods of the plugin and copies them in a separate external class (e.g., `external_class` in Step 1). Then, in Steps 2 and 3, *MARVEL* replaces the body of the selected methods (i.e., the `or_code`) with some fake instructions (i.e., in our example, a new `RuntimeException`) and it compiles each extracted method in an external DEX file. At the end of the procedure, there are as many distinct DEX files as the number of removed methods. The goal of the fake instructions is to break the flow and the logic of an app, avoiding this being recognizable by an analysis tool or a reverse engineer expert; thus, the more reasonable and varied the bogus code is, the greater the resilience to analysis techniques is.

The DEX files (i.e., the `external_class`-es) are not saved in the plugin APK, as they are directly dispatched to the trusted container through a secure channel (e.g., downloaded in an encrypted form from a remote, trusted server) during the installation of the plugin. At runtime, the container restores the original code of the removed methods only on-demand (i.e., before their execution). Thus, an attacker should extensively execute the protected app

to trigger all the removed methods, retrieve their external DEX files, and reconstruct the original app's logic.

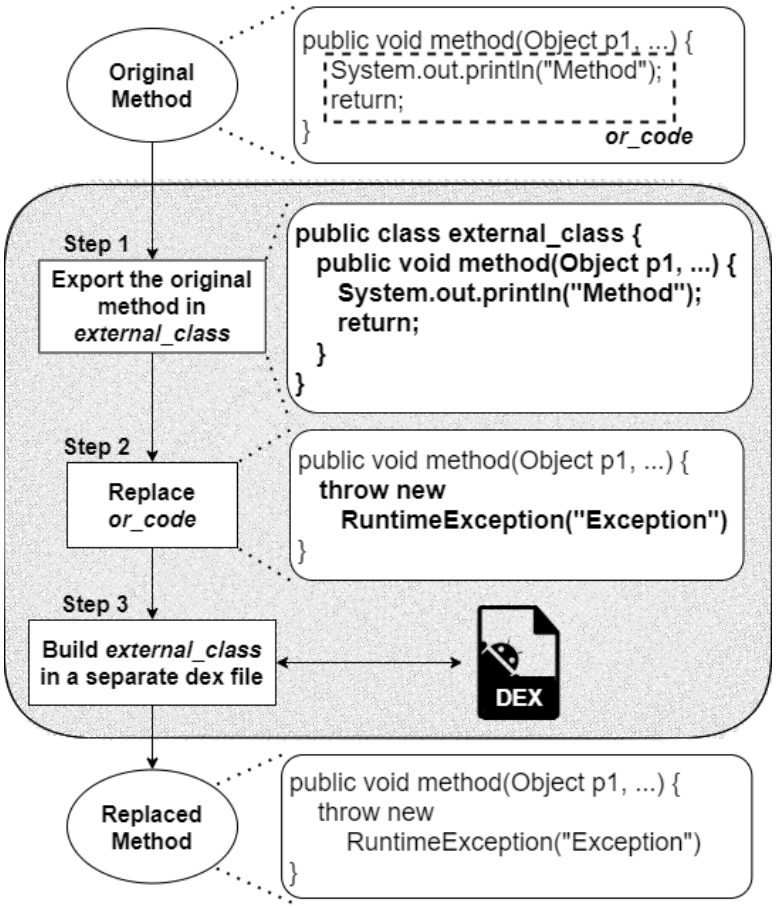


Figure 3.4 The *MARVEL* code splitting transformation process.

3.5.3 Interconnected Anti-Tampering Controls

To address both the *R2* and the *R3* requirements, *MARVEL* extends the traditional anti-behavioral controls (e.g., APK tampering verification) techniques to protect both the container and the plugin from repackaging attacks. In particular, we designed interconnected anti-tampering checks, which are novel integrity control systems based on a challenge-response protocol between the container and the plugin. The container leverages its intrinsic role to work as a proxy between the plugin and the Android operating system.

MARVEL supports two different types of IAT: *base IAT* and *IAT with encryption*. In the former, the plugin verifies the response from the container against a set of pre-computed

hashes. In contrast, in the latter, the plugin uses the response from the container as a decryption key for a ciphered portion of its code.

Base IAT allows the injection of controls, which are less CPU-demanding than IAT with encryption (i.e., hash verification versus code decryption). Moreover, the presence of two IATs increases the entropy of the protected APK since the attacker has to detect and deactivate two different types of control. Finally, it is worth noticing that the extracted code of a code-splitting technique can hide one or more IATs, further increasing the difficulty of a successful repackaging attack.

Figure 3.5 shows the transformation steps in creating an IAT with encryption. In particular, *MARVEL* first generates a challenge-response pair (Step 1). Then, it replaces the original code of the method with its encrypted form generated using `R` – the response to the challenge – as the encryption key (Step 2). The procedure injects the challenge vector (i.e., an Android API such as `getPackageInfo`) in the body of the method (Step 3), which contains the challenge value (`C`) as a parameter. Finally, *MARVEL* adds the logic to perform the decryption with the response value sent by the container and the execution of the decrypted code thanks to the Dynamic Code Loading (Step 4).

Figure 3.6 depicts the injection process for a base IAT. Unlike the previous case, *MARVEL* does not encrypt the `or_code`. Furthermore, in Step 3, the procedure adds the logic to verify the response obtained by the container against a pre-generated hash of R (i.e., $HASH_R$). Similar to logic bombs, the protection of a base IAT is granted by the one-way property of cryptographic hash functions, which makes it extremely hard for the attacker to retrieve the original value (i.e., the response) from its hash.

MARVEL stores the pre-generated pairs of challenge-response for a plugin app in an external resource (e.g., a JSON file), which needs to be delivered to the trusted container at the first execution of the plugin.

3.5.4 Runtime Execution

Figure 3.7 summarizes the *MARVEL* architecture and the interactions between the trusted container and the protected plugin sample(s). At the startup of a plugin app, the container performs some APK tampering verification checks over the plugin app to verify some of its metadata (e.g., the signature of the plugin APK file). If the checks fail, the container does not execute the plugin. In the same way, the plugin carries out, during its execution, an evaluation of the running virtual environment (i.e., anti-virtualization-like controls), for instance, by checking the class name of the proxy object injected by the container. If this check fails, the

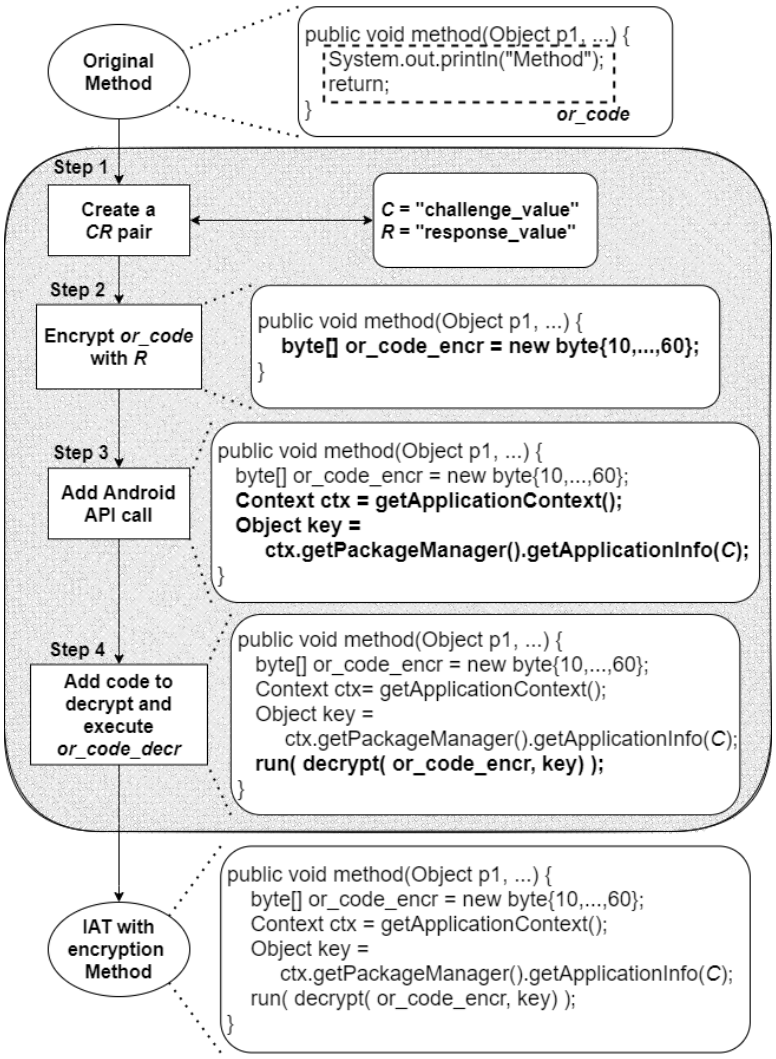


Figure 3.5 The *MARVEL* IAT with encryption transformation process.

app crashes immediately. Such integrity controls enable a further detection of a tampered environment, preventing the runtime execution of the plugin in an untrusted - or malicious - container. Afterward, the *MARVEL* container manages the additional anti-repackaging controls, i.e., code splitting and IATs.

Code Splitting. At runtime, the container detects whenever the plugin loads a fake method (i.e., a method with bogus code) and replaces it with the correct one. To this aim, the container injects a Custom ClassLoader (CCL) in the plugin process to intercept the loading of any new class in the current DVM memory. Once the loading event occurs (e.g., method M_i in Figure 3.7), the CCL resolves the requested class, and if it contains a fake method, the CCL injects the original method's code (i.e., retrieved from the DEX file DEX_i) in the plugin

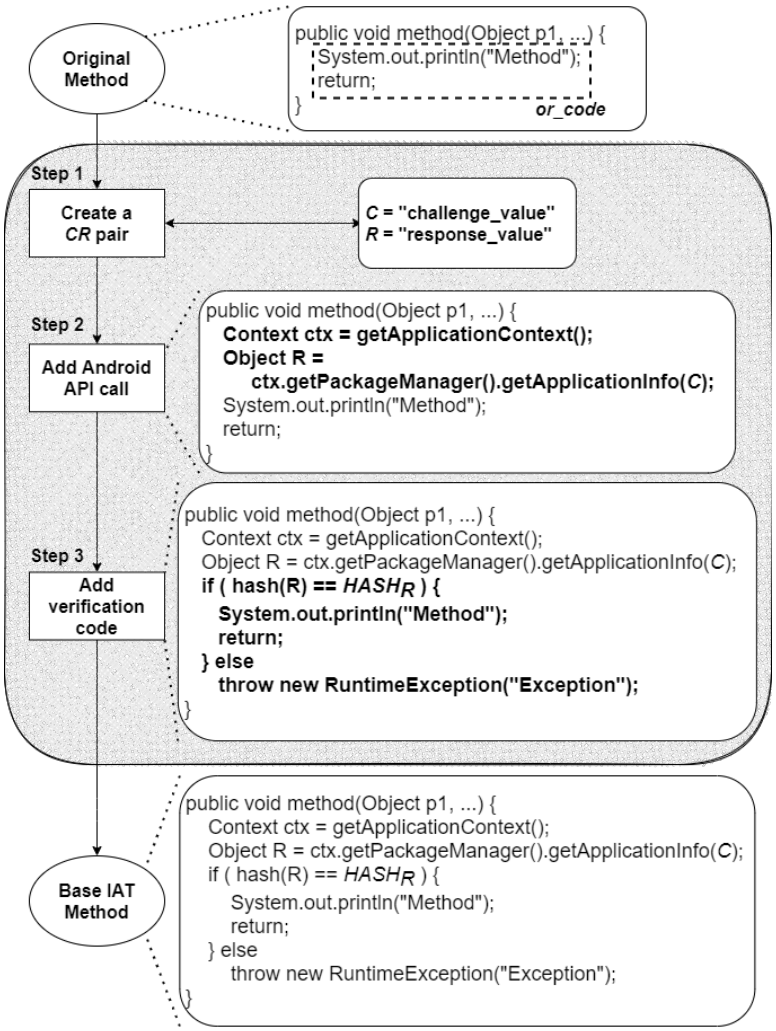


Figure 3.6 The *MARVEL* base IAT transformation process.

memory instead of the fake one. It is worth noticing that this operation is transparent to the plugin, which receives the resolved method (i.e., CM_i).

IAT Controls. When the plugin executes any IAT, it first calls an Android API with the challenge value inside the request (e.g., `getApplicationInfo(C)`). The proxy module of the container intercepts the API call and searches for any challenge value in its CR pairs list. If a match is found, the container does not forward the API invocation to the Android OS and replies to the plugin with the corresponding response value (`Res[C]`).

Upon receiving the response, the plugin executes the IAT control. In the case of a base IAT, the app verifies if `hash(Res[C])` is equal to the pre-computed hash ($HASH_R$) and if this is not the case, the IAT raises a security exception and terminates its execution. In case of an IAT with encryption, the plugin uses `Res[C]` as the key to decrypt `or_code_enc`. If

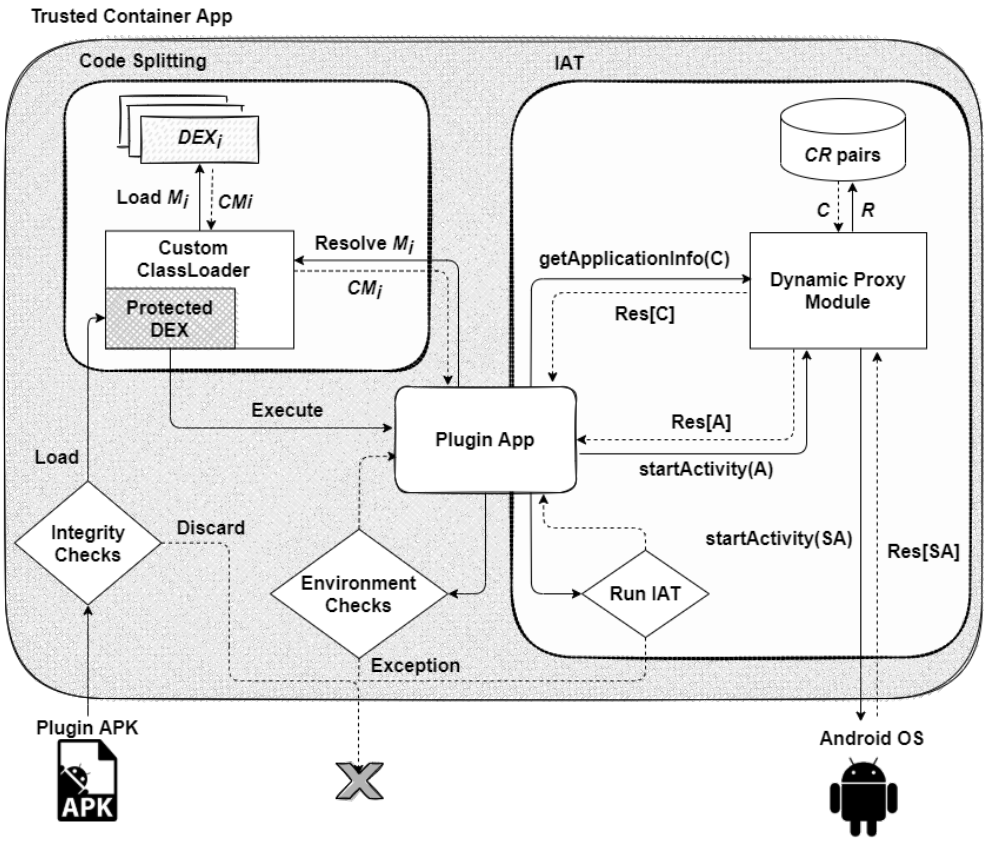


Figure 3.7 The *MARVEL* architecture.

the key matches the expected value (i.e., *R*), the plugin can execute the original method; otherwise, (if the decryption fails), the IAT terminates its execution and throws a security exception.

3.6 Implementation of *MARVEL*

We implemented the *MARVEL* protection scheme in a prototype tool called *MARVELoid*, which injects the protection in the plugins. Furthermore, we extended the VirtualApp framework [139] to develop the trusted container responsible for enforcing the runtime protection. It is worth highlighting that, given the standard design, *MARVEL* can also be ported on DroidPlugin (or, in general, other app-level virtualization frameworks). The source code of *MARVELoid* and the trusted container are available at [225].

3.6.1 *MARVELoid*

MARVELoid is a Java-based tool devoted to generating the protected plugin starting from the original app. The tool leverages the Soot framework [108] to analyze and modify the bytecode of the Android app and on Jarsigner [159] to sign the output APK file with a valid certificate.

MARVELoid requires three input parameters, namely the probability to I) replace a method through code splitting (P_{repl}), II) inject a base IAT control (P_{base_IAT}), and III) inject an encrypted IAT (P_{enc_IAT}). Overall, the probability of introducing a protection is given by:

$$P_{protection} = (1 - P_{repl})(1 - P_{base\_IAT})(1 - P_{enc\_IAT}),$$

which is calculated as the product of the mutual independent probabilities that refer to the non-introduction of each protection.

Also, *MARVELoid* requires the package name (PN) that identifies the part of the app's bytecode (i.e., the group of classes) to include in the protection process. The default value of PN is the app's package name, which is specified in the Manifest file of the APK.

Given an app, *MARVELoid* unpacks the APK file to extract the DEX file(s) (i.e., the `classes.dex` files) and translates the Java bytecode into an intermediate representation (i.e., Jimple) by leveraging the Soot framework. Then, the tool scans the code and, for each method, computes the probability of protecting it according to the input percentages. Thus, for each method, *MARVELoid* calculates the chance to inject each protection (i.e., code splitting, base IAT, and IAT with encryption). To this aim, it creates three objects that extend the `BodyTransformer` class of the Soot framework. These objects execute concurrently and compute the probability of protecting a method with their specific protection type; in this way, according to the input probability, a method can be protected with multiple protection types in a random order, depending on the scheduling of the objects.

Code Splitting. Concerning the code splitting procedure, *MARVELoid* replaces the body of a plugin method with a fake one. In detail, the tool creates a new `external_class` with a method that has the same signature as the original one (i.e., the same parameters and return type) and the same logic (i.e., instructions from the original method are removed and injected into the new one). The new class is then saved into a DEX file, while a new body replaces the original one of the plugin; in our implementation, we injected a throw statement of a `RuntimeException`. *MARVELoid* stores all the created DEX files (one for each replaced method) outside the original APK to be delivered to the trusted container. In this way, if the

plugin is executed outside the trusted container, it crashes as the execution of the replaced method throws an exception.

IAT Controls. To inject an IAT into a method of the plugin, *MARVELoid* first generates a challenge-response pair (i.e., `C` and `R` values) of 32-bit each and stores them in an external JSON file.

In the case of a base IAT, the tool injects a call to the `getPackageInfo` API, adding the value of the challenge into the request. Then, it includes the code to verify the response value against the hash of the pre-generated challenge value.

In the IAT with encryption, instead, *MARVELoid* extracts the body of a method (i.e., `or_code`) and encrypts it using the AES256 ECB algorithm and the `R` value as the encryption key. The result is saved into a byte array (i.e., `or_code_encr`), and the tool injects a call to `getApplicationInfo` API, adding the `C` value into the request. Finally, *MARVELoid* adds the code to decrypt and execute `or_code_encr`. *MARVELoid* keeps track of each challenge-response pair to guarantee that the container has the information required to compute the response to a specific challenge.

Moreover, regardless of the IAT type, *MARVELoid* could also inject evasive controls to enhance the robustness against static and dynamic analysis.

At the end of the protection process, *MARVELoid* recreates the `classes.dex` file(s), builds the APK, and re-signs it with a valid certificate. Also, it extracts the metadata information of the resulting APK (e.g., the sha256 hash and the APK signature) required by the container to perform the integrity checks on the plugin.

3.6.2 Trusted Container

The trusted container is an extension of the VirtualApp framework in charge of offering the virtualized environment to execute the plugins and enforcing the *MARVEL* protections. In detail, the container I) performs the integrity checks of the plugin, II) interacts with them through the IAT, and III) restores the original content of the fake methods (for the code-splitting mechanisms).

In our prototype, the container retrieves the protection data of the plugin (i.e., metadata, challenge-response pairs, and the set of external DEX files) directly from the device's internal memory. However, the container may download that information from a remote and trusted server in a real scenario.

Code Splitting. To keep track of the loading of a fake method, the container injects a custom ClassLoader (CCL) into the plugin process. The CCL is an instance of the `dalvik.-`

`system.DexClassLoader` and overloads the `loadClass` method. In particular, the CCL I) resolves the requested class, II) checks if it contains some fake methods, and III) in case of fake methods, it replaces them with the correct ones.

The container leverages the DCL (i.e., Dynamic Code Loading) to load the original instructions into the plugin process and ART instrumentation [24] to inject the correct bytecode into the method's memory. To implement the ART instrumentation inside the trusted container, we leveraged the YAHFA hooking framework [166]. With the support of YAHFA, we can overcome the limitations of the Java Dynamic Proxy that restricts the hooking only to object instances of classes with at least one interface. In detail, ART instrumentation allows the modification of the `ArtMethod` object instance that the DVM exploits to manage Java methods internally. The container overrides the value of the `entryPointFromQuickCompiledCode`, which is the pointer to the compiled code of the interpreter (i.e., the DEX file of the plugin), with a pointer to the loaded code of the replaced method (included in a DEX file during the *MARVELoid* process).

IAT Controls. The container scans all Android API calls from the plugins to detect the challenge values. The trusted container leverages the Java Dynamic Proxy to modify the proxy object of the Android Package Manager. From a technical standpoint, we replaced the proxy class generated by VirtualApp for the `getPackageInfo` API (used by the base IATs) and `getApplicationInfo` API (used by the IATs with encryption) with our custom logic. In particular, the container verifies the content of the request. If it matches a challenge, the container retrieves the correct response value and returns it to the plugin instead of forwarding the request to the Android framework. *MARVELoid* supports the configuration of multiple APIs to deliver the challenge of the IATs and thus could be customized.

3.7 Experimental Evaluation

We empirically assessed the performance of the *MARVEL* methodology by I) applying *MARVELoid* over a dataset of $4,000$ apps. The dataset's apps were downloaded from the Google Play Store in December 2020, have an average rating of 4 stars on the Google Play Store, and belong to more than 30 categories. The experimental campaign aims to evaluate I) the distribution of the protection values according to several combinations of input percentages (i.e., *static evaluation*), and II) executing a subset of 45 randomly selected apps (with the best sets of input parameters) to evaluate their compatibility with virtualized environments and the introduced runtime overhead (i.e., *runtime evaluation*).

Static evaluation. The experiments were hosted on a virtual machine running Ubuntu 20.04 with eight processors and 32GB RAM. All experiments were conducted using the default value of the PN input parameter of *MARVELoid* (i.e., the package name of each app). We conducted tuning tests to detect the on-average best combinations of input percentages (i.e., P_{repl}, P_{enc_AT}, and P_{base_IAT}) to ensure a reasonable trade-off between the protection overhead values (i.e., space and time overhead) and the protection level (i.e., the number of injected protections). In particular, we computed the protection overhead and protection values for 24 combinations (i.e., the permutations of $[5, 10, 20, 30]$).

MARVELoid was able to apply $96,000$ protections (i.e., 4k apps for each permutation) in nearly 109 days and 14 hours (i.e., 9477578 seconds) of computation; thus, the protection of a single app took, on average, 98 seconds.

MARVELoid worked successfully in 97.3% of the cases (i.e., cases in which it generated a valid protected APK). At the same time, the remaining 2.7% (i.e., 2624) failed due to well-known bugs of the adopted libraries (such as a Soot bug [127]) or crashes during the tool transformation process.

Figure 3.8 shows the distribution of the success percentage for each permutation. Although the value slightly decreases with higher values of P_{repl}, it is worth noticing that its range always sits between 96% and 98%.

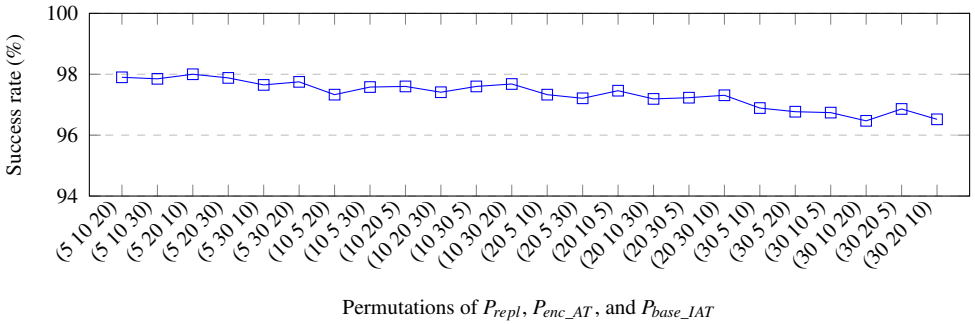


Figure 3.8 Percentage of apps successfully protected by *MARVELoid*.

Then, we calculated the average percentage of space overhead of the protected APKs compared with the original ones in each permutation. Figure 3.9 shows the results for the 24 combinations of the input values. The value distribution suggests that the space overhead is directly proportional to P_{enc_IAT}: such behavior is reasonable because *MARVELoid* adds some auxiliary code to handle the decryption and execution of the encrypted IAT. In the worst case, the space overhead introduced by the protection is always less than 18% of the original APK size.

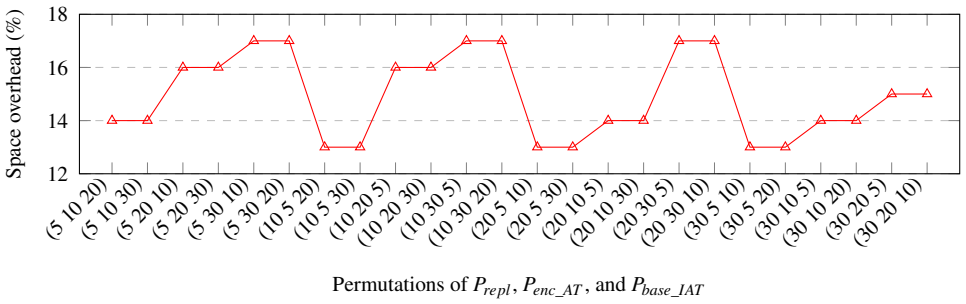


Figure 3.9 Space overhead introduced by *MARVELoid*.

Finally, Figure 3.10 shows the average number and distribution of the injected protections (i.e., the average number of replaced methods, IAT with encryption, and base IAT, respectively) for each permutation of the input probability values. During the experimental activities, *MARVELoid* was able to inject a minimum of 68 and a maximum of 135 protection elements in the protected apps regardless of the input parameter permutation.

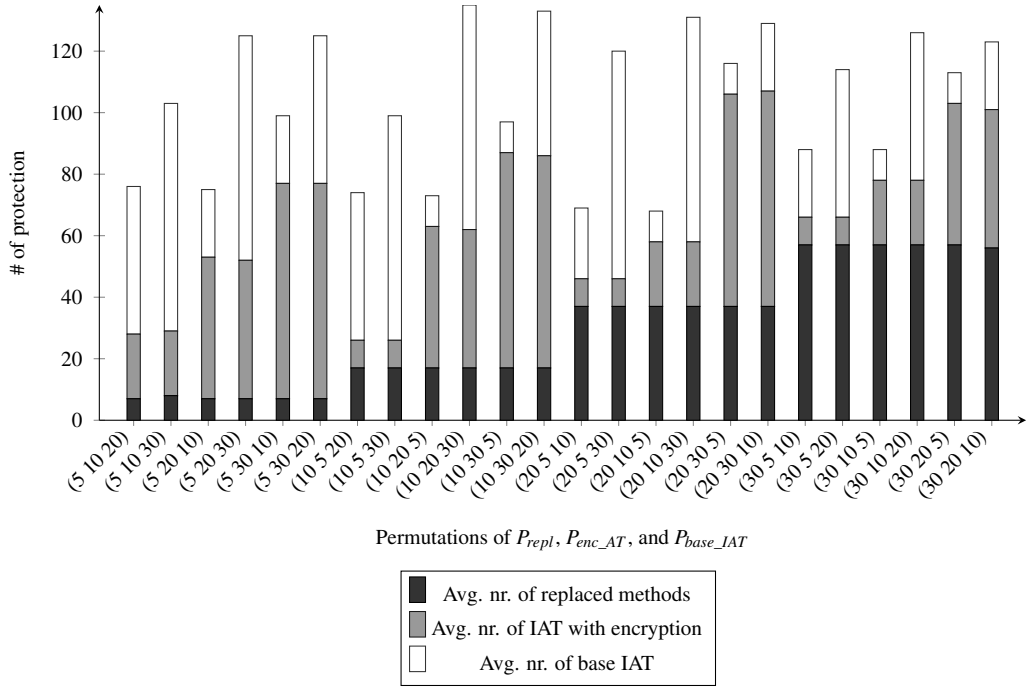


Figure 3.10 Distribution of the average protection values inserted by *MARVELoid*.

Runtime evaluation. The last set of experiments is aimed at evaluating the effectiveness and usability of the proposed protection scheme. To this aim, we randomly selected 45 apps from the dataset of 4k apps that reported at least $5,000,000$ downloads on the Google Play Store in December 2020.

In the first part of the runtime evaluation, we assessed the compatibility of traditional Android apps w.r.t. virtualization and the overhead introduced regarding CPU and memory usage. Then, we evaluated I) the effectiveness of *MARVELoid* by running the protected apps inside the trusted container and II) the corresponding overhead (using the results of the standard virtualization as reference values).

For the testing phase, we used a set of emulated Android 8.0 devices equipped with a dual-core processor and 2GB of RAM and ARES [181]. This black-box tool leverages Deep Reinforcement Learning to stimulate Android apps automatically.

To evaluate the compatibility with the virtual environment, we installed and executed the 45 apps directly on the emulator to check their proper functioning and resource usage. Then, we ran the apps in a standard VirtualApp container; in this step, we collected the overhead introduced by the Android virtualization in terms of CPU and memory usage. At the end of this phase, we identified five apps that triggered an exception due to the virtual environment. Thus, we discarded such apps from the rest of the experiments.

Then, we applied the protection to the remaining 40 apps with two different combinations of input percentages. To define the best combinations, we computed the average number of methods inside the default package name of the apps. Then, we calculated the parameters that allowed injecting at least 50 protection elements in each app. Being 600 the average number of methods per app in the dataset, the combinations are $P_{repl} = P_{base\_IAT} = P_{enc\_AT} = 10\%$ ($Setup_{10\%}$) and $P_{repl} = P_{base\_IAT} = P_{enc\_AT} = 15\%$ ($Setup_{15\%}$).

The protection phase introduced, on average, 111 protections (i.e., 32 replaced methods, 38 IAT with encryption, and 41 base IAT) for $Setup_{10\%}$ and 175 protections (i.e., 51 replaced methods, 58 IAT with encryption, and 66 base IAT) for the second setup. Moreover, we executed every app ten times for each combination for 4 minutes to verify its correct execution inside the trusted container and the introduced overhead.

During the experiment, 22 out of 40 (with $Setup_{10\%}$) and 25 out of 40 (with $Setup_{15\%}$) apps executed successfully. The remaining apps (i.e., 18 and 15) threw new exceptions, crashed, or became unusable. We manually investigated such problems to discover the following causes:

- 8 apps ($Setup_{10\%}$) and 7 apps ($Setup_{15\%}$) crashed due to a well-known Soot bug [21, 224]. The bugs affect the transformation process from the bytecode to Jimple of third-party libraries included in APK files, causing the final app to crash during the execution.

- 4 apps ($Setup_{10\%}$) and 4 apps ($Setup_{15\%}$) triggered a timeout defined by the UiAutomator2 library, used by ARES to test the application. Unfortunately, after a deeper investigation, we discovered that such timeout is hardcoded and cannot be modified [155]. It is worth noticing that these apps did not crash due to the protection but were terminated by the library.
- Finally, six apps ($Setup_{10\%}$) and four apps ($Setup_{15\%}$) threw new exceptions directly related to the protection process.

Concerning the last point, the root cause analysis of the crashes caused by *MARVELoid* led to the discovery of two issues of our prototype implementation. The first one is related to an incorrect referencing of the Context object of the app; in several cases, the plugin raised a null pointer exception due to the absence of the correct reference. The second bug occurs if a replaced method is loaded in the CCL. In such a case, a crash can occur because the CCL does not inject the correct body of the method, causing a runtime exception.

		Simple container	$Setup_{10\%}$	$Setup_{15\%}$
CPU	min.	-1.1	-0.4	-0.3
	avg.	+0.9	+1.7	+4.7
	st.dev.	1.3	2.7	4.6
	max.	+10.9	+7.6	+24.5
Memory	min.	+3.8	-0.1	-0.5
	avg.	+4.4	+0.2	+0.2
	st.dev.	1.9	0.1	0.2
	max.	+6.1	+0.6	+0.7

Table 3.1 CPU and memory usage overhead in percentage point (pp).

The dynamic testing phase allowed us to monitor the overhead introduced by traditional virtualization and our solution on 45 apps. Table 3.1 shows the minimum, maximum, average, and standard deviation of the overhead values expressed in percentage points (pp). A positive value means that the apps used more resources than the reference value, while a negative is the vice versa. Moreover, a value of X means that the app used X more or less (depending on the sign) resources in that environment compared to the reference value. In the first column, we compared the execution of apps in the virtual environment with the traditional execution method (i.e., the app directly installed on the actual device). For the protected apps (i.e., second and third columns), we computed the overhead compared to the execution in a clean

virtual environment. In this way, we can discriminate the virtualization overhead from the one introduced by our protection scheme. Since the execution of an app under virtualization is composed of two processes (i.e., container and plugin), the overall amount of CPU and memory is given by the sum of the overhead of these two processes.

The average overhead the virtualization introduces is negligible regarding CPU usage (i.e., an increase of 0.9 pp). On the contrary, the memory overhead is 4.4 pp. Also, the negative minimum value for the CPU overhead denotes that the virtual environment app can adopt CPU optimization strategies at the cost of a higher memory footprint.

The analysis for the protected apps with $Setup_{10\%}$ shows that the average overhead introduced by the protection is negligible, i.e., an increase of 1.7 pp for the CPU usage and 0.2 points for the memory. In the worst-case scenario, the CPU overhead reached 7.6 pp. We advocate that such an increase is caused by the overhead required by the CCL and ART instrumentation to load and inject the fake methods.

The results on the memory overhead are confirmed by the analysis of the protected apps with $Setup_{15\%}$. In particular, on average, the memory overhead is negligible, with a peak of 0.7 percentage points in the worst case. On the contrary, the CPU overhead is 4.7 pp, reaching, in the worst case, an increase of 24.5 pp.

3.8 Discussion

From a security point of view, *MARVEL* is effective. Concerning the repackaging attacks presented in Section 3.4.1, *MARVEL* provides preventive anti-repackaging protection in both repackaging attack scenarios (i.e., traditional and through virtualization).

First of all, the analysis of the victim app APK (i.e., Step 1 in Figure 3.1) becomes more challenging for the attacker due to the code splitting and the IAT with encryption, which removes or encrypts some portions of code from the protected app. Concerning the identification and neutralization of the repackaging protections the victim app might be equipped with (Step 2 and Step 3 in Figure 3.1), the mutual collaboration between the trusted container and the plugin makes this goal harder to achieve for the attacker: she has to investigate the runtime communication between the TC and the plugin through dynamic analysis techniques, to retrieve the *CR* pairs or the original code after code splitting. Moreover, since a method can be concurrently transformed by several protection mechanisms (e.g., an extracted method can contain an IAT), the attacker has to recursively resolve the nested protection once she has disabled the external one. Finally, the customization of a container to set up a virtualization-based repackaging attack (Step 2 and Step 3 in Figure 3.3) is prevented

by *MARVEL* thanks to the mutual integrity checks between the trusted container and the plugin, which stops the execution of the latter in case the environment is detected to be not secure. Overall, *MARVEL* makes virtualization-based anti-repackaging attacks as complex as traditional ones.

Chapter 4

The Dark Side of Native Code on Android

4.1 Introduction

Since its very first version, Android has supported Java Native Interface (JNI), a mechanism to connect the Java language, which typically runs inside a virtual machine, and C/C++ languages, which are instead compiled into native libraries. While using JNI by benign apps has brought numerous advantages in performance and resource consumption, it has also introduced multiple challenges when used by malicious software. While apps written in Java are not dependent on the device's architecture, the same cannot be said for the native components that use JNI. This, combined with the fact that today the Android system can run on many architectures, introduces numerous challenges for a malware analysis pipeline. Moreover, JNI serves a variety of different purposes. For instance, it can interact with the Android runtime and instantiate new objects or modify their fields. It can also perform low-level operations or be used as a trampoline to jump back to the ART. This versatility and the additional complexity of its analysis have led malware authors to increasingly use the native layer as a protection mechanism to hide malicious code, perform suspicious operations, or complicate static and dynamic analysis [174, 235, 236]. In addition, benign actors leverage native code to implement more robust protection techniques; for instance, in [148], authors exploit the compiled code to implement more sturdy APK tampering verification controls.

Currently, JN-SAF [236] and JuCify [188] represent state-of-the-art solutions to analyze apps with native code to detect data leaks statically. However, these tools only focus on detecting data leaks through Java and native code. In addition, they rely on Angr's symbolic execution [230], which often incurs a path explosion that prevents it from completing the analysis. However, despite the native code's growing popularity, no previous work has documented how Android malware uses and abuses the native layer. Furthermore, current

Android anti-malware engines pay little attention to the native components. For instance, we created and submitted to VirusTotal some malicious samples with well-known publicly available exploits in the native code: half of the samples went undetected, and the remaining were detected by just one engine.

This chapter presents a new approach to studying Android malware to fill this gap. It provides a methodology to reverse engineer Android apps that use native code, considering all suspicious patterns that can be used for malicious purposes. The methodology is implemented in *ANDani*, a framework to detect and *tag* suspicious and surreptitious native code usage in Android apps. These tags are handy because they can be imported into reverse engineering frameworks and highlight portions of (native) code that the analyst needs to focus on, as they may conceal malicious behaviors.

We decided to follow a different approach w.r.t. the state-of-the-art, and we developed our analysis infrastructure by combining two components. The first is an extended version of the Soot framework for bytecode analysis; we improved the entry points detection and managing concurrent execution. The second is an architecture-independent Ghidra plugin for the native code analysis that can handle JNI data structures and propagate inferred types from JNI methods' signatures through various functions.

To study the evolution of surreptitious native code patterns over the years, compare the use of native functionalities in benign and malicious apps, and understand the underlying motivation behind possible discrepancies, we analyzed with *ANDani* a total of $113,476$ APKs that include native code. Such APKs are divided into two different datasets: $97,829$ malware from AndroZoo [12] spanning from 2010 to 2021, and $15,647$ benign apps from the most downloaded apps of the Google Play Store.

In particular, the measurement aims to answer the following five main research questions.
RQ1: For what purpose do malicious apps use native libraries?
RQ2: Which operations are hidden in the native layer?
RQ3: How has the usage of native code in malware evolved?
RQ4: How do malware and goodware differ in using native code?
RQ5: Is native code information useful in the "war" against malware?

The results led to numerous insights into using native components: malware is more likely to trigger the native component without user interaction, especially during the app startup and while reacting to Broadcast Receivers. For example, waiting for the mobile phone to be charged can indicate the right time for malware to perform intensive operations that would typically lead to excessive battery consumption and raise the victim's suspicion. Native components are often employed for obfuscation by dynamically loading and executing

Java and other native code. In particular, the malware goes native to dynamically load and invoke methods of the Android framework that require dangerous permission to get the user's sensitive information in a twisted way. This creates leaps between these two different code "worlds", making the analysis particularly difficult. Moreover, we also report an undocumented strategy: malware exploits JNI mechanisms to load malicious or harmless code at will, depending on evasive controls. Such new trends are bridging different functions between the Java world to the native one or including two libraries with the same name but different functionality. Finally, we found that malicious native libraries are often re-used among samples for several years without the authors changing their hash.

The measurement study also highlighted differences in how benign and malicious apps use native code. To prove the usefulness of *ANDani* and the reliability of the tags that can be extracted from its output, we used them to train and test a Random Forest algorithm to classify goodware and malware. Our results are promising: the classifier can distinguish between the two classes, leveraging only the native features with an average error of 0.02 and achieving an F1-score of 0.97. The output of the classification task allowed us to discriminate suspicious behaviors that are more correlated to malware and whose presence can hint at potential malicious patterns, thus providing valuable information to the malware analysts. For example, we found that how native components are triggered through JNI contributes nearly 50 percent of the accuracy.

In summary, this chapter makes the following contributions:

- a detailed methodology to assist analysts in reverse engineering native code by Android apps, which focuses on the disclosure of suspicious (and potentially harmful) operations;
- *ANDani*, a static analysis tool for Android apps to perform an in-depth behavior analysis of all aspects related to the native code;
- the first longitudinal analysis on Android malware, specifically focusing on native code, over the past ten years and in current 'top apps' on the Google Play Store to investigate the security impact of the native code and understand its behavior;
- novel anti-analysis techniques, against both static and dynamic analysis, that we found in malware and goodware;
- a concrete use case of suspicious tags usage: it obtained remarkable performances in a binary classification task as a sole feature.

4.2 Android JNI Internals

This section discussed the details of the native code usage in Android apps. In particular, the Android system allows an app to invoke and use native code through four primary techniques, whether in the form of shared objects or executable files.

4.2.1 Native Library Loading

To allow the interaction between Java code and native components through JNI, native libraries (in the form of SO files) must first be loaded into the app's address space. An app can load these libraries by using the `load` or `loadLibrary` methods, which are present in both `java.lang.System` and `java.lang.Runtime` classes. For both implementations, the difference between the `load` and `loadLibrary` implementations is that the first method requires the library name to be specified as an absolute path. In contrast, the second requires that the name passed as an argument not contain a file extension or path, as the library will be automatically searched in the default path under the installation folder (see Section 2.1.1).

When the library is loaded, the linker calls the initialization functions. The ELF file format defines three sections that contain code (or pointers to code) that are in charge of initialization procedures: `.pre_initarray`, `.init`, and `.initarray`. The linker searches them in this order and runs the code of the present ones. In Android, the `.pre_initarray` section is ignored for shared libraries [103]. Finally, the linker invokes a JNI-specific initializer, the `JNI_OnLoad` function.

4.2.2 Bridging Functions

Since the JNI allows the interaction between Java and native components, it is possible to invoke a native function defined within a shared object from a Java method. Vice versa, the native component, always via JNI, can interact freely with the Java counterpart. For example, the native component can create objects, invoke methods of the Android framework or defined within the app itself, or even modify field values: all these operations are possible thanks to the use of *JNI Callbacks*.

In the Java code, the methods declared with the keyword `native` represent the functions defined and exported within the native shared libraries accessible from the app. The redirection of the execution flow and the mapping between the native method definition and its implementation are all handled via JNI. In particular, when a native library is loaded, the JNI

tries to resolve the native methods dynamically and map them into the corresponding defined Java method [163].

These steps are possible thanks to the fixed structure in the *naming convention* of the native methods. The name of the function, translated from Java to native, is made of three parts: the "`Java_`" string, concatenated with a mangled fully-qualified class name of the related Java class, and the name of the method. For instance, in the following example

```
1 package xx.yyy;
2 class Clazz {
3   public native String test(int x);
4 }
```

the class `Clazz` declares a native method `test`. When the shared library containing the function is loaded, the system will search for the symbol corresponding to the function name: `Java_xx_yyy_Clazz_test(JNIEnv*, jobject, jint)`.

Furthermore, the definition of native methods requires the first additional argument always to be a pointer to `JNIEnv`. Then, in the case of a static method, JNI requires the second argument to be a pointer to the corresponding Java class (i.e., an object of type `jclass`); on the other hand, a pointer to the corresponding Java object (`jobject`). The first and the second arguments are implicit, and the developer does not directly handle them.

The third and last argument of the example, the type `int` defined in the Java method signature, matches the native type `jint`. For a complete list of Java primitive types and their machine-dependent native equivalents, please refer to [160].

The `JNIEnv` type – a `struct` when the shared object is written in C, or a `class` in C++ – contains pointers to functions that allow the interaction between the native component and the Android framework or the app itself. These functions are called *JNI callbacks*, and the most relevant are `NewObject`, `FindClass`, `GetMethodID`, `GetStaticMethodID`, and the `Call*` family (e.g., `CallVoidMethod`). Through these functions, the native code can respectively instantiate objects, find a reference to a class or a method, and then call a method.

Moreover, the `JNIEnv` provides the `RegisterNatives` function to dynamically map a Java method defined as `native` to its implementation in the shared library at runtime. However, following a fixed naming convention is not required in this case. For a complete listing of all the JNI functions, please refer to [162].

4.2.3 Native Activity

Developers who require their app to have high performance in terms of execution speed or that need to interact with low-level system components may decide to develop *the entire app natively*. For this, the NDK introduces and supports the concept of "Android Native Activity". The native code implements the Android activity component, and its methods are invoked according to the activity lifecycle functions (e.g., `onCreate`, `onDestroy` [85]).

If the app does not contain any Java code, a (Java) "stub" is created at compile-time with the sole task of loading and running the native code since it is released as Shared Objects and, therefore, must follow the loading process described above.

There are several requirements for the developer to create a native app: it must target an API level greater than eight and specify whether it contains Java code via the `android:hasCode` attribute of the Manifest. Then, each Activity defined as native must indicate in which library it is located: the name of the shared library is specified in the `android:name` attribute of the activity tag in the Manifest file.

It is worth noticing that the Activity is the only component that can be fully developed in native code. The others (i.e., Services, Broadcast Receivers, and Content Providers) must be defined in Java, and then they can interact with the native counterpart.

4.2.4 Process Execution Methods

Shared Objects are not the only types of ELF that can be executed within Android apps. The Android framework allows apps to execute shell commands, scripts, or ELF executables in a separate process through the `Runtime` class, with its `exec` methods, or via the `ProcessBuilder` class and its `start` method. These methods allow the app to execute binaries that do not contain any JNI components. The execution of these new processes takes place in a different process, and therefore, the interaction between the native component and the app is not handled by JNI.

4.3 Motivation & Methodology

In the first part, this section shows a practical example of the limitations of the current antivirus engines in analyzing native code. Then, it defines what "suspicious" means and illustrates the proposed methodology guided by an example, highlighting how the native layer can be abused for malicious purposes.

4.3.1 Native Components and Antivirus Software

To begin with, this section shows how the static module of many Android anti-malware engines completely ignores the presence of native components or performs only fundamental controls. In this respect, we collected well-known exploits (CVE-2011-1823 [41, 215], CVE-2014-3153 [42, 215], CVE-2016-5195 [43, 117], and CVE-2019-2215 [44, 34, 120, 54]) from public repositories from 2015, 2016, and 2019. It is worth noting that those do not simply contain a proof-of-concept for a given vulnerability but working exploits. Moreover, anti-malware engines are aware of these exploits; for example, the malicious app `cdde`[1] is labeled with the exploit name or the associated CVE (i.e., CVE-2016-5195 is also known as *DirtyCow*).

Then, we created a native (x86 and ARM32) Android app for each exploit. Each app loads and runs the exploit immediately at startup: it loads the library in the static constructor of the `Application` class. It calls the function running the exploit directly from `JNI_OnLoad`, making the program flow to reach the exploit straightforward and trivial to analyze. The apps were signed with the default debug key and did not use any form of obfuscation or shrinking. The resulting six apps were uploaded to VirusTotal and scanned with at least 61 engines. At the time of the analysis, three apps were detected by only one engine (i.e., samples `7383`, `e088`, and `1f26`), and we obtained no detections for the remaining three (i.e., samples `5bd3`, `858e`, and `f7b9`). Since we used only publicly available and well-known exploits (with no changes), our experiment reveals an explicit limitation in the existing commercial solutions to which this chapter hopes to contribute.

4.3.2 Suspicious Pattern

The proposed methodology comprises seven different *Steps* to analyze native-code execution flows and identify suspicious patterns effectively.

We define a *suspicious pattern* as any use of native code that shows characteristics of surreptitious code [153] (e.g., obfuscation and anti-tampering). It is worth emphasizing that suspiciousness does not necessarily imply maliciousness. Such techniques typically prevent others from exploiting the intellectual effort invested in producing software, regardless of whether the action is for uncovering malicious purposes. Therefore, when using the word 'suspicious,' we refer to a code snippet that needs further investigation without binding our definition to a particular technique or malicious behavior.

[1] Due to the ease of readability, this thesis reports only the first four bytes of the apps' sha256; while the complete list is in Table A.1 in the Appendix.

4.3.3 Running Example

Listing 4.1 JNI example – Java side

```
1  package com.xmp;
2  public class MainActivity extends AppCompatActivity {
3    static {
4      System.loadLibrary("xmplib");
5    }
6
7    public native boolean fun1(int a, int b);
8    public static native String fun2(Object obj, boolean b);
9
10   public String getSubscriberId(){
11     return "no_id";
12   }
13
14   @Override protected void onCreate(Bundle b) {
15     super.onCreate(b);
16     ActivityMainBinding bind =
           ActivityMainBinding.inflate(getLayoutInflater());
17     setContentView(bind.getRoot());
18     String s = fun2(getSystemService(Context.TELEPHONY_SERVICE),
           fun1(2, 1));
19     bind.sampleText.setText(s);
20   }
21 }
```

To better present our methodology, this section continues the discussion guided by the code given in Listing 4.1, which shows an example of an Android app that displays a string to the user in a blank Activity. The string is generated [line 4.1.18] from two native functions, `fun1` [line 4.1.7] and `fun2` [line 4.1.8] implemented in the `xmplib` library (loaded at [line 4.1.4]). The C code (reported in Listing 4.2) declares three functions: `JNI_OnLoad` [lines 4.2.22-28], `Java_com_xmp_NatClass_fun1` [lines 4.2.1-4] and `sensitive` [lines 4.2.6-16]. The native functions reachable from the Java code are those that respect the JNI *naming convention* [163] or the ones that are registered through the `RegisterNatives` callback. In particular, the `JNI_OnLoad` binds at runtime, through the `RegisterNatives`, the Java method `fun2` to the native function `sensitive` [line 4.2.26]. Instead, the function `fun1` leverages the JNI name convention to register it and returns a boolean depending on comparing its two integer parameters. Lastly, the `sensitive` function, depending on the value of its boolean parameter, uses `FindClass` to get the reference to the user-defined `MainActivity` class or the `TelephonyManager` of the Android framework, and, finally, invokes the `getSubscriberId` method of such class. In this example, `fun2` is called with

Listing 4.2 JNI example – Native side

```
1  extern "C" JNIEXPORT jboolean JNICALL
2  Java_com_xmp_MainActivity_fun1 (JNIEnv* env, jobject thiz, jint a,
       jint b){
3    return a>b;
4  }
5
6  static jstring sensitive(JNIEnv* env, jclass jclazz, jobject obj,
       jboolean b) {
7    char* cName;
8    if (b)
9      cName = "android/telephony/TelephonyManager";
10   else
11     cName = "com/xmp/MainActivity";
12
13   jclass fclazz = env->FindClass(cName);
14   jmethodID method = env->GetMethodID(fclazz, "getSubscriberId",
         "()Ljava/lang/String;");
15   return (jstring) env->CallObjectMethod(obj, method);
16 }
17
18 static JNINativeMethod nat_methods[] = {
19   {"fun2", "(Ljava/lang/Object;Z)Ljava/lang/String;",
         (void*)sensitive },
20 };
21
22 jint JNI_OnLoad(JavaVM* vm, void* reserved) {
23   JNIEnv* env = nullptr;
24   vm->GetEnv(reinterpret_cast<void **>(&env), JNI_VERSION_1_4);
25   jclass clazz = env->FindClass("com/xmp/MainActivity");
26   env->RegisterNatives(clazz, nat_methods, 1);
27   return JNI_VERSION_1_4;
28 }
```

a `True` argument (because `fun1` with such arguments returns `True`) to obtain the IMSI via `getSubscriberId`. However, more complex flows could dynamically load a DEX file that calls `fun2` passing `False` as an argument, thus making it possible for the `getSubscriberId` of the `MainActivity` to be invoked.

4.3.4 Anatomy of the Analysis

The execution order of runtime code is crucial to preserve when reverse engineering an Android app that uses the JNI. To closely follow the execution flow, we have broken down our analysis pipeline into the seven main steps in Figure 4.1.

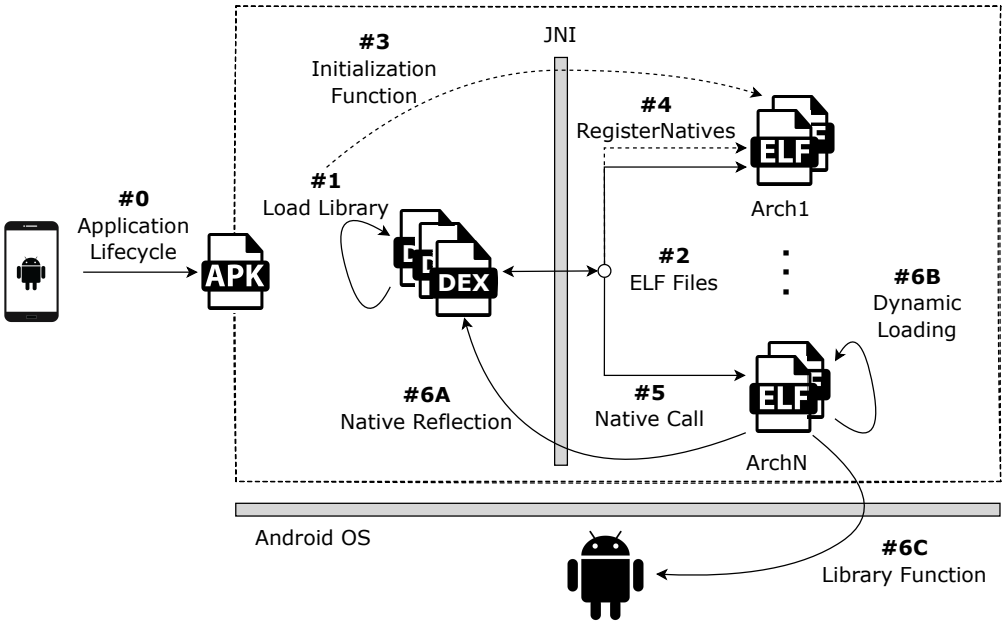


Figure 4.1 JNI behaviour

The very first step, Step **#0**, extracts the possible entry points of the app from its Manifest file to understand the Android components (i.e., Activity, Service, Broadcast Receiver, Content Providers) and their lifecycles. Such information is fundamental to identifying all the app components that can be triggered and could potentially reach the JNI. For example, malware can gain persistence by registering a Broadcast Receiver for the `BOOT_COMPLETED` intent filter to start each time the device boots and loads the malicious native component. In the running example, the `onCreate` method of the `MainActivity` [line 4.1.14] class is an entry point.

In Step **#1**, the Java code loads a native library. Calls to load methods mainly occur in the class's static constructor containing the native methods (e.g., line 4.1.4). It is worth recalling

that the `load` method (of both classes `java.lang.System` and `Runtime`) allows the code to load a native library from the filesystem, not necessarily located in the default folder. In contrast, the `loadLibrary` requires that the name passed as an argument does not contain a file extension or path, as the library will be automatically searched in the default path. From a malicious perspective, the `load` method can hide which ELF file is loaded. This type of analysis can reveal (e.g., sample `04CE`) whether a native library is not present in the APK but is downloaded at runtime once specific conditions are met (e.g., after some evasive controls) or whether the app loads a file that is not supposed to contain executable code (e.g., loads a PNG file as ELF file). Moreover, this approach can uncover the author's will to hide the library loaded by the app if the string passed as an argument to the load method is not defined in the code but computed at runtime.

In Step **#2**, JNI automatically loads the correct library according to the device architecture. However, malware can ship libraries with the same name, exposing and implementing functions with different names or semantics. For instance, if malware authors knew that specific antivirus solutions run the app in x86 emulators, they could avoid detection by restricting the malicious logic to the ARM architecture and placing a harmless code into the x86 library. The malicious component would evade the analysis since the sandbox only loads the x86 library. However, when executed on an actual device that supports ARM, the malware would show its actual behavior.

In Step **#3**, the dynamic linker first invokes the initialization functions of the ELF file (e.g., `.init_array`), and then, JNI automatically calls the `JNI_OnLoad` function (e.g., line 4.2.22). A malicious actor can hide the logic to perform evasive checks in the initialization functions. For example, we found samples (e.g., `019E`) using the `ptrace` function as an anti-debugging technique when it is loaded in memory (i.e., the check is done in the `JNI_OnLoad` method).

Furthermore, in Step **#4**, the `JNI_OnLoad` is used to dynamically link the JNI methods through the `RegisterNatives` API (e.g., line 4.2.26). In this way, the mapping between Java methods and native functions is not statically explicit anymore but is performed at runtime. Thus, an attacker can perform environment checks and use the `RegisterNatives` to map different functions depending on their results (e.g., `7900`, discussed in Section 4.6).

At this point, in Step **#5**, the Java methods can call the native functions of the loaded library (e.g., line 4.1.18). This transition is crucial as it is impossible to determine the mapping between methods and functions statically and as different architectures result in different semantics. For this reason, we designed three specific Steps (**#6A**, **#6B**, **#6C** in no particular order) to be followed once the execution moves from Java to the native library.

Step **#6A** tracks the *native reflection*, which allows native code to manage Java objects through the JNI callbacks (e.g., create Java objects, invoke Java methods, or modify object's fields). For example, the `sensitive` method of the Listing 4.2 [lines 4.2.6-16] uses the JNI callbacks `FindClass`, `GetMethodID`, and `CallObjectMethod` to get a reference of the object `TelephonyManager` and call its method `getSubscriberId`. This possibility significantly complicates the analysis because it makes it only possible to statically determine which code will be executed by resolving the arguments of such methods. In addition, it is also worth noting that the native reflection can also use Java reflection features to hide methods or accessed fields. For instance, sample `4E4B` uses the `getDeclaringClass` method of the `java.lang.reflect.Method` class to dynamically retrieve the class representing an object. Then, it leverages the native reflection (e.g., `FromReflectedMethod` callback function) to retrieve the corresponding method from the reflected object.

In Step **#6B**, the native code can dynamically load and invoke exported functions of other libraries by relying on the `dlopen` and `dlsym` functions. This technique aims to conceal the usage of a particular shared library from static analysis, given that it is no longer present in the dependencies. Even if the system should prevent loading or linking those libraries since Android 7, we found multiple samples (e.g., `1306`) that use dynamic loading to load and call functions from a library not present in the APK.

The last step to consider, Step **#6C**, is using library functions that can affect security, for example, running exploits against specific subsystems or performing environment checks (e.g., debugger detection). We divided them into nine categories: Dynamic Loading (e.g., `dlopen`), Execution (e.g., `system`), File Permission (e.g., `chmod`), Kernel Interaction (e.g., `ioctl`), Identity (e.g., `geteuid`), Memory Protection (e.g., `mprotect`), Network (e.g., `gethostbyname`), Open Special File (e.g., `open("/proc/version")`), Process Management (e.g., `ptrace`), and Monitoring (e.g., `inotify_add_watch`). Table 4.1 reports the complete list.

4.4 Suspicious Analysis Framework

This section describes *ANDani*, a cross-architecture analysis framework that implements our methodology. The first part presents a general overview of the framework, and then the section describes each module in detail.

Table 4.1 Security-relevant library calls

Category	Library Calls
Dynamic Loading	dl(v)sym, dl(m)open
Execution	exec*, system, popen
File Permission	*chmod*, *chown*, access
Kernel Interaction	ioctl, syscall
Identity	get(e)uid, get(e)gid
Memory Protection	mmap, mprotect
Network	socket, listen, connect, gethostbyname
Open Special File	*open* `<special_file_path>`
Process Management	kill, ptrace, fork
Monitoring	inotify_*

4.4.1 Overview

ANDani receives as input an APK file. It first unpacks and extracts its DEX, JAR, and ELF files. This operation recursively occurs on all the archives (e.g., ZIP, TAR) inside the APK. It also parses the Manifest file to extract relevant information, such as the Android components, the required permissions, and the various intent filters.

Then, *ANDani* starts the analysis by computing an Inter-Procedural Control Flow Graph (IPCFG) for both the Java and the native code. First, it computes the IPCFG of every code file, then merges them into a single IPCFG, keeping track of whether the code file was found in a standard or non-standard location within the APK. This information is crucial to identify malicious code in non-standard locations (e.g., in an archive file).

The IPCFG is based on two types of nodes: *code blocks* that are the traditional basic blocks interrupted by function calls, and *call blocks* that represent the function calls or method invocations. Moreover, our analysis also considers that from a call block of a Java native method, we can have multiple edges to different native functions of different architectures. In the same way, when we deal with the native reflection (see Step #6A of Section 4.3) where the function can take multiple arguments (e.g., lines 8-15 of Listing 4.2), the graph can also have different edges to Java methods.

The *Bytecode module* handles the IPCFG for all the bytecode components, and it is built on top of the Soot framework [226, 108]. For the native components, the analysis is performed by the *Native module*, which leverages the Ghidra [79] API to process the ELF files.

4.4.2 Bytecode Module

The Bytecode module performs the analysis of DEX and JAR files. It is written in about 1814 lines of Java code and based on the Soot framework. The module starts by translating the bytecode into Jimple, a three-address code [7] intermediate representation that Soot needs to build the preliminary IPCFG.

Soot suffers from many known limitations in the case of parallel and asynchronous Java classes, i.e., those that extend or implement `Thread`, `Runnable`, `AsyncTask`, or `Timer`. Such drawbacks make the IPCFG incomplete; therefore, we had to add the necessary code – complete with test cases – to handle them.

Once the IPCFG is computed, our system continues the analysis by identifying the calls to the *load* methods that allow loading native libraries. Each time a call is found, it tries to resolve the argument (a string) to identify which native library is loaded. For this, we perform a backward intra-procedural taint data analysis. This is sufficient in most cases because, as previously mentioned, calls to load methods almost always occur in the class's static constructor containing the native methods, which also references the plaintext string with the library's name.

The system then repeats this procedure for each DEX and JAR file, and at the end of this phase, it produces the IPCFG of the whole bytecode found in the APK. Then, the analysis identifies the entry points of the app by combining the information from the Manifest file (e.g., the `onCreate` method of the main Activity) and the list of entry point methods of previous studies [22, 234]. As the last step, the system identifies all native methods and extracts their signatures, which serve as input to the next module.

Example of Thread handling. Listing 4.3 shows a simple Java snippet where the `run()` method of a Thread subclass is executed through the `Thread.start()` call. The code contains two Thread classes, respectively `MyThreadA` and `MyThreadB`, but only the first one is used. The *Bytecode module* has been designed to consider the call's context and propagate the arguments. In this example, contrary to other state-of-the-art tools, the bytecode module can create an edge from `Main.start_thread` to `MyThreadA.run` method, ignoring the `MyThreadB` class.

Listing 4.3 Executing a new sample thread in Java.

```
1  class MyThreadA extends Thread {
2    void run() {
3      System.out.println("My Thread A - called");
4    }
5  }
```

```
6
7       class MyThreadB extends Thread {
8         void run() {
9           System.out.println("My Thread B - never called");
10        }
11      }
12
13      class Main {
14        static void start_thread(Thread t) {
15          t.start();
16        }
17
18        static void main(String[] argv) {
19          start_thread(new MyThreadA());
20        }
21      }
```

4.4.3 Native Module

The Native module is written with almost 5200 lines of Java code, and it uses the Ghidra reverse engineering framework API to perform headless analysis of all ELF files. The analysis leverages the Ghidra *P-code* intermediate representation to model the behavior of many architectures. Since the signatures of the Java native methods are fundamental to propagating types of information in the native code properly, this module is executed after the Bytecode module. The output of the Native module is the final, complete, and merged IPCFG.

The analysis performed at this stage can be divided into three different phases.

I) ELF information and JNI entry point identification. Given an ELF file, the module extracts generic information from the ELF header (e.g., architecture, sections, segments, symbols, strings), and it initializes the set of the JNI entry points, i.e., the native functions that can be reached by Java code. To start, we consider as JNI entry points the functions whose symbol name respects the JNI naming convention – i.e., symbol name starts with `Java_` or `JNI_` – and all ELF initializers. If we consider the example shown in the previous section, the entry points are: `JNI_OnLoad` [line 4.2.22] and `Java_com_xmp_MainActivity_fun1` [line 4.2.2].

II) Entry point arguments type definition. The module iterates the entry points: for `JNI_OnLoad` and ELF initializers, if present, it creates an edge from the respective Java load

method to the first of these being called, and the others are propagated. For the `Java_`-functions, given the signatures of Java native methods collected from the Bytecode module, it creates an edge from the corresponding Java method. It applies the proper JNI data type to all the input parameters. If the corresponding `Java_` function is not found, it just applies the `JNIEnv*` type to the first one. Once the argument types are updated in the JNI entry point, the module propagates them to the called functions. If we consider our initial example, in that case, we have two edges: the first one from the `loadLibrary` [line 4.1.4] to the `JNI_OnLoad` [line 4.2.22] and the second from the Java method `fun1` [line 4.1.7] to the `Java_com_xmp_MainActivity_fun1` native function [line 4.2.2].

However, in case the native functions are dynamically registered through the `RegisterNatives` (e.g., [line 4.2.26]), the function names may not start with `Java_`, as it is the case in our example for the `sensitive` function. To handle these cases, the module searches for all `RegisterNatives` calls in the code of the entry points, and, for each of them, it performs a backward taint data analysis again on its second and third arguments. The second argument of the `RegisterNatives` is a `jclass` object obtained from a call to the method `FindClass` of the `JNIEnv`. The `FindClass` takes a string with the corresponding Java class name; it is marked as tainted and the string is searched backward. The third argument is a `JNINativeMethod` object that contains the mapping between the Java methods and native functions. Lastly, after this procedure, the module re-iterates the JNI entry points to apply the correct data types. This phase is crucial to get the correct types, especially `JavaVM` and `JNIEnv` since they expose all the JNI callbacks.

Following the example, after correctly resolving the arguments, the module can now create an edge from the Java method `fun2` [line 4.1.8] to the `sensitive` native function [line 4.2.6] and apply the correct type to its arguments.

III) Graph construction. At this last stage, since all the entry points and the types of their arguments have been retrieved, the module can create and merge the final IPCFG. This is done by following the call blocks with a depth-first strategy, propagating the types of arguments to the called functions each time. However, we are not just interested in JNI callbacks but also specific Android system library calls, as their arguments can reveal helpful information about the app's capabilities and its suspicious behavior.

Once the module finds an argument of interest, it marks it as tainted, and it performs a recursive backward taint analysis on the tainted arguments to resolve them. For example, *ANDani* would try to resolve the path of each opened file to detect suspicious accesses to Linux special files, such as `/proc/version` to retrieve the version of the Linux kernel [123] (sample `0259`). For more complex cases, such as the one in our running example in which

the argument is not unique [lines 4.2.8-11], this part of the analysis considers all the values that the variable can take (among those that were able to recover). In the example, the graph construction would add two edges from the `CallObjectMethod` to the `getSubscriberId` method: the first to the `TelephonyManager` class, and the second to the `MainActivity` class.

4.4.4 Suspicious Tags

At the end of the analysis, the tool visits the IPCFG, and if it finds a match with one of the patterns described in our methodology, it assigns a tag to the corresponding node where the violation occurs. A tag comprises the concatenations of two strings (the category and the title) with the symbol "-". Referring to Section 4.3.4, the categories are each of the steps presented in Figure 4.1, while a title is a short description of a suspicious pattern of the methodology.

Table B.1 in the Appendix reports all the tags. To give an example, a node tagged with `NR_FINDCLASS-JAVA_REFLECTION` (category: "`NR_FINDCLASS`", title "`JAVA_REFLECTION`") denotes that *ANDani* found a call to the `FindClass` callback (native reflection), that it was able to resolve the argument, and such argument refers to a Java class related to reflection.

These tags serve three purposes. First, they can assist a human analyst by showing precisely which parts of the code need to be inspected because they might hide, for example, some evasive technique, malicious behavior, or protection mechanisms. Second, it provides automated analysis systems with target locations that could be investigated using more costly but precise analysis routines (e.g., symbolic execution or dynamic analysis). Finally, the suspicious tags can improve classification tasks, revealing the most distinctive features of malware samples, as shown in Section 4.7.

4.5 Dataset

To perform our analysis, we built a comprehensive dataset of Android apps, divided into *malware* samples collected over the past ten years and a *goodware* dataset of benign apps. All malicious samples are downloaded from AndroZoo [12]. In addition to the APK file, AndroZoo also provides: I) the date associated with the APK file, II) the number of antivirus (AV) engines that detected the app as malicious on VirusTotal (VT), and III) the date on which the app was submitted to VT. However, according to their documentation, most apps

from Google Play have 1980 as the APK date. Therefore, we assigned each app to a year by applying the following procedure: if a year was present and had a plausible value, i.e., other than 1980 and between 2010 and 2021, we consider that to be the year. Otherwise, we assign the year of the app as the year in which the first scan in VT was performed. On the other hand, benign apps were collected among the most downloaded apps from the Google PlayStore until April 2021 for each of the 50 categories [96]: 15 categories are related to *Games* while the remaining 35 vary from *Communications* to *Social*.

Since this research focuses on the usage of the native component via JNI by Android apps, our dataset consists only of apps that make use of this technology. Therefore, each app in our dataset respects at least one of the following two constraints: it must contain a DEX file with a declaration of at least one `native` method that is not defined in the standard Android libraries, or it must contain an ELF Shared Object file with a JNI entry point method.

Malware. First, we considered "malware" all samples with at least five AV detections. Since Androzoo does not indicate the family the samples belong to, we downloaded the respective report from VirusTotal and determined the family via AVClass2 [192]. From a preliminary analysis, we found that the eligible samples, grouped by year, are overrepresented by a few families (e.g., among 10k random APKs, 41% of the malicious samples in 2014 belong to just three families). Therefore, we opted to group samples by pair of years, with a maximum of 5% for each family, and the sha256 hash of each sample belongs to just a couple of years (namely, the intersection between the samples for each pair of years is empty).

We ended up with $97,829$ native malicious apps, whose distribution is summarized in Table 4.2, where we report the percentage of the three most frequent families. It is worth noting that the number of malware that AVClass2 cannot assign to a family (*Singleton*) has increased over the years. This is because the number of AV engines has increased, and there are more inconsistencies in the naming convention of family labels; furthermore, we observed that in the Androzoo dataset, the average number of detections (also by referring to updated VirusTotal reports) in recent malware is lower than the old one.

Goodware. We collected the package names of the 500 most downloaded free apps for each of the 50 official Google Play Store categories. We extracted this information using *Google Play Scraper* [61], and we downloaded the samples with *Playstore Downloader* [78]. The tool was able to download $27,665$ apps successfully. After our pre-filtering, which only retained apps that use native components, we were left with $15,647$ samples. The fact that more than half (57%) use a JNI component is a clear sign that, nowadays, the native layer constitutes a fundamental part of the Android userspace ecosystem.

Table 4.2 Distribution of the suspicious samples taken from AndroZoo

	Detection Range				
Year	**Fam. 1**	**Fam. 2**	**Fam. 3**	**Singleton**	**Total**
2010/11	5.0%	5.0%	5.0%	8.4%	5,065
2012/13	5.0%	5.0%	5.0%	7.0%	15,458
2014/15	5.0%	5.0%	5.0%	16.5%	19,127
2016/17	5.0%	4.8%	4.4%	37.3%	20,268
2018/19	5.0%	5.0%	4.3%	48.5%	19,610
2020/21	3.7%	0.5%	0.5%	84.3%	18,301
Total:					97,829

4.6 Results

This section presents and discusses the results of the longitudinal measurement conducted over the malware and the goodware datasets and answers to the first set of research questions. To better understand why malware uses, and in particular abuses, native code and how it is tied to the app's lifecycle, the results will be reported according to the seven main steps of our methodology (discussed in Section 4.3.4). Moreover, at the end of this section, the work summarizes the main takeaways that the measurement revealed concerning using native code in malicious and benign samples. Since malware is grouped into pairs of years, we will refer only to the highest year (e.g., 2011 refers to the pair 2010/2011) to improve readability.

***ANDani* Performances.** On our machine, an Ubuntu 20.04 with 64 CPUs (Intel Xeon 8160 @ 2.10GHz) and 128 GiB of RAM, we measured an average execution time of 614 seconds (std. dev. 182) for a single APK. Such variance is because the analysis duration of *ANDani* grows proportionally with the code (i.e., ELF or DEX files) in the APK.

4.6.1 App Lifecycle

The first question to answer with this measurement is *when* apps invoke JNI methods, i.e., whether the native component comes into play immediately after the app starts or is only invoked when specific conditions are met.

For each of the native functions, *ANDani* first visits the Java IPCFG to verify if a native method is *reachable* (a method is considered to be reachable if there is at least one block of the IPCFG that calls such method) and, if it is, it extracts all the possible entry points of the different paths that lead to it. This first analysis shows that among goodware, 91% of the native functions are reachable from nearly all (99.8%) of the DEX files in the standard

location. Thus, the code invoking the native component is easy to identify, not obfuscated or dynamically loaded. Moreover, it is essential to highlight how 94% of these functions are reachable only under specific *user interaction* with the app, such as a click on a GUI item. Among the remaining, 4.4% of the native functions are reachable from the lifecycle methods of Activity components, and the remaining (1.2%) is triggered at the app's startup, from static constructors, or other Android components (i.e., Service, Broadcast Receiver, and Content Provider).

The picture is utterly different for malware. In fact, among malicious apps, the average number of reachable native functions has decreased from 81% in 2013 to 21% in 2021. Moreover, the number of reachable native code only from DEX files not located in the standard position is always higher than 2%. These results suggest that malware uses resources loaded at runtime to invoke native methods. Furthermore, since 2017, more than 34% of the native functions have been reachable at the startup (i.e., directly from the Application class), with a peak of 53% in 2021. This observation is significant because it shows how malware, unlike goodware, tries to start the native component as quickly as possible. Comparing the 2021 percentages for both malware and goodware, we observe that for the malicious apps, 55.4% of the entry points are Application (53%) or Activity (2.4%) lifecycle methods, and only 44% are related to user interaction. This again highlights that current malware mainly invokes native code at the beginning of the process and does not wait for the user to interact with the app.

Another interesting aspect that differentiates the use of alternative entry points used by goodware and malware concerns Broadcast Receivers. Although their use is very limited in percentage, our data shows that malicious apps are more prone than goodware to use broadcast receivers to invoke native functions when they are notified that the user is present, an existing app has been added or removed from the device, an external power has been (dis)connected to the device, or the device boots.

The final analysis measured when an app declares Java native functions that shared libraries do not expose and vice versa, namely exported JNI entry points not declared in the Java code. Our results show that these discrepancies are much more prevalent in malware than in goodware. For instance, in 2017, 49% of malicious apps exported JNI entry points were never declared in the Java code, and it grew over the years until 84% in 2021. In goodware, this behavior only appears in 21% of the apps. One possible explanation, confirmed later in this section, is that malicious apps dynamically load Java code from native and then use this new Java code to invoke other exported native functions. This redirection cycle between Java and native layers is a form of obfuscation that complicates static analysis.

4.6.2 Load Methods

Android apps can load native libraries through the `load` and `loadLibrary` methods. We recall that the former accepts the full path of the library, while the latter takes only the library's name, which is loaded from the default folder.

The 86% of loading operations in the goodware dataset load shared libraries directly from the standard location using the `loadLibrary` method. Until 2013, this was also the preferred method among the malicious samples, with more than 89% of such operations relying on `loadLibrary` and only 11% on the `load` method. From 2013 to 2021 instead, this percentage steadily decreased, and today, the `load` method accounts for more than 42% of the loading operations against 14% in goodware.

A second crucial aspect is when these libraries are loaded to make the native methods accessible. The JNI common practice suggests loading the libraries within the static constructor so that native methods will be immediately available and exposed to the rest of the app's methods. We measured this from 2011 to 2015: more than 69% of malware was loading libraries from a static constructor. However, from 2017, we observed a change that saw samples loading libraries in other points of the app's execution until 2021 when 80% of the malware loads native libraries from other code locations. Goodware reinforces this phenomenon by loading libraries from different entry points, but still, almost half of the loading operations are performed by static constructors.

Other exciting aspects of the Application Lifecycle analysis concern user interaction and response to specific system events. In the first case, the experiments show that 52% of goodware loads native libraries only in response to user interaction, while malware performs this behavior only in 16% of the samples. Moreover, malware tends to use broadcast receivers to load native libraries in response to particular events, such as an external power has been (dis)connected to the device, an external media is (un)mounted, or the device boots.

Finally, we analyzed the names of the libraries that goodware and malware load. Our data shows that, throughout the years, malware loads significantly more native libraries related to packing (e.g., `jiagu` [1]), obfuscation, encoding/encryption, or audio recording. On the other hand, a high percentage of goodware includes libraries from well-known frameworks, such as Unity [218] and Flutter [68].

Table 4.3 Supported architectures by goodware and malware sets over the years [%]

	ARM 32-bit	ARM 64-bit	x86	x86_64	MIPS 32-bit	MIPS 64-bit	Others
Good '21	80.49	81.26	59.32	43.27	11.18	6.71	0.15
Mal '11	99.98	1.68	7.65	0.68	4.05	0.10	0.00
Mal '13	99.99	0.33	8.92	0.13	2.59	0.04	0.01
Mal '15	99.99	3.52	38.42	1.77	12.29	1.53	0.00
Mal '17	99.99	30.93	65.49	22.86	17.86	9.97	0.18
Mal '19	99.96	46.42	69.98	29.31	8.90	5.36	0.50
Mal '21	99.54	78.51	87.97	45.69	2.19	1.20	1.43

4.6.3 ELF files

To support different architectures, APKs include ELF libraries in `lib` folder, which are divided into different subfolders, one for each architecture supported by the app. Table 4.3 details the evolution of the architectures supported by malware and goodware.

Google Play introduced the *App Bundle* [86], a new publishing format to generate and serve optimized APKs for each device configuration. This ensures that new apps that use native libraries do not have to ship their APK with multiple versions of the same native library since the correct version will be directly shipped at installation time. Despite this new feature, we observed that 75% of the goodware still ships the same library object for multiple architectures, and more than 11% still includes MIPS – even though it is no longer supported [95]. In comparison, almost all malware samples include ELF files for ARM32, and the number of samples with multiple architectures grew over the year, from 9% in 2011 to 74% in 2021. Since 2017, malware samples also contain few ELF files that target different architectures (e.g., `SPARC` and `PowerPC`), which grows up to 1.4% in 2021. We also observed a small percentage (about 0.1% every year) of ELF files with a broken header section, while none of the goodware had this peculiarity. By inspecting some of these cases manually, we identified a common technique to prevent static analysis: the authors have removed a part of the ELF header from the libraries, and the missing part is only restored at runtime.

Afterward, we analyzed where the ELF files are located in the APK. Most of the goodware (78%) contain ELF files only in the standard `lib` folder, while 10% ships ELF files only in non-standard (e.g., `assets` folder or archives) locations. On the other hand, about 85% of the malware contains ELF files in a non-standard location, which reflects the high usage of the `load` method in malware. Moreover, malicious APKs also embed native libraries in archives or try to disguise the analyst by hiding the executable with a wrong and harmless extension name (i.e., different from `.so`). In particular, more than 5% of ELF files in the

malware samples were extracted from an archive or had an extension that did not match the file type. PNG, JAR, SDK, and LIB are the most common wrong extension names.

We also found another interesting phenomenon: since 2017, more than 1% of malware contains multiple ELF files with the same name that target the same architecture in the standard folder and a non-standard location. This phenomenon is almost negligible among goodware (less than 0.1%). Therefore, we manually examined those ELF files with the same name and architecture. We computed the Jaccard similarity coefficient between the set of JNI entry points and obtained an average index of 0.7 when there should be no difference. It indicates that the two files have very different entry points. For instance, sample `213c` contains an ARM library in both the `lib` and the `assets` folders. The ELF library under the `assets` folder exports one entry point more than the one in the `lib` folder, and such an entry point is in charge of executing a well-known exploit [215] to escalate privileges and obtain root capabilities. In general, from 2015, more than 6% of such libraries (with a peak of more than 30% in 2019) implement the `JNI_OnLoad` entry point only in the non-standard location.

Finally, we searched how many ELF files are shared among different malware samples and found that the percentage of native libraries shared between at least two malicious APKs is 46% ($60,895/131,325$). Among the top 1,000 shared ELF files, 59 (almost 6%) have more than four detections, and some have up to 41 detections in VirusTotal. Interestingly, among the top shared ELFs in a non-standard location (e.g., `asset` folder), more than 10% has a VT detection higher than five. Moreover, 7% of the shared ELFs are contained in different places depending on the malware. It highlights how malware tends to include the same malicious executables not in the `lib/` folder, reinforcing the results of the previous sections that malware leverages the `load` method to get libraries from various locations. For instance, the `6c6e` ELF file is shared among 293 APKs and loaded from the `res/` folder. We then measured the years these malicious ELFs were observed in our dataset and obtained a median of four years. The most extreme case is `d867` shared by 54 APKs from 2011 to 2021 and has 32 detections on VT. As it turns out, malware authors are not too concerned about antivirus software and do not even try to change the hash of the malicious components they keep reusing year after year.

Last, our data shows that 6% of the ELF files are shared between at least one malware and one goodware sample. This highlights how malware provides goodware-like features to fool the final user (e.g., repackaged apps) or goodware developers try to protect their code with the same obfuscation techniques of malware apps.

4.6.4 Initialization Functions & JNI_OnLoad

When the dynamic linker loads a native library, it first calls the initialization functions defined in well-known ELF sections. Samples in both datasets – malware and goodware – have the same percentages of initialization functions distributed across sections, while the `.init` section is used by less than 1% on average, `.init_array` is more prevalent, being present in roughly 30% of the apps.

After the ELF initialization functions are executed, JNI calls the `JNI_OnLoad` function. An exciting trend is related to the fact that while the usage of `JNI_OnLoad` in the goodware dataset is around 60%, the percentage of malware that uses it had been steadily increasing over the years until 2017, after which it remained stable at 92%. This brings another important observation: `JNI_OnLoad` is the function where usually the `RegisterNatives` is used to register native methods dynamically, thus hiding the mapping between Java and native functions. Therefore, malware may use this construct more frequently to prevent the detection of which native methods are executed by the sample. During the years, the percentage of such malware that exports only the `JNI_OnLoad` as JNI entry point decreased from 48.1% to 18.1%, while, for goodware, it is stable at around 42%. This shows that malware authors jointly use both techniques to register the JNI entry points to hide the mapping for specific functions. Moreover, we measured the average number of branches of the `JNI_OnLoad`: for malware, it grew up over the years until 91 (std. dev. 131) in 2021, while, for goodware, it is only 44 (std. dev. 110); a clear sign that malware authors often abuse this function and must be analyzed with particular care.

ANDani tries to resolve which methods are dynamically registered by computing the arguments of the `RegisterNatives` calls. The usage of this callback in malicious apps has increased over the years up to 71% of samples, while it accounts only for 30% of the goodware samples. We recall that the `RegisterNatives` dynamically maps a Java method defined as `native` to its implementation in the shared library at runtime, accepting two distinct input parameters: the Java class and the function method mapping. *ANDani* resolved the mapping in 88% of the cases for benign samples, thanks to goodware often using hardcoded values. For malware, the percentage decreases to 78% averaged over the years as a consequence of the fact that these samples tend to obfuscate the parameters' value. Conversely, resolved Java classes are approximately 60% for goodware and 55% for malware since 2019. For the arguments we were able to resolve, we inspected whether the related classes were defined within the code of the APK, the framework, or if they were not present in either. We identified how the number of Java classes used in the `RegisterNatives` not present in the APK is higher in malware. In goodware, 78% of the resolved classes

point to defined references (and all the other points to Android framework classes, such as `androidx.renderscript.RenderScript`), while for malware since 2019, this only accounts for 36% of the classes. The remaining resolved classes (64%) for malicious samples were not in any DEX file.

We also noticed that malware samples perform various checks (e.g., environment verification controls, anti-debugging checks, etc.) and map different functions depending on their results. For example, we identified one APK (`7900`), which loads a native library (`32cd`), and it leverages this technique to invoke the `RegisterNatives` callback and to map different Java methods depending on the context in which it is analyzed. A more detailed manual analysis revealed that the sample decrypts a string, and if a class with the same name exists, it maps the native functions to such class; otherwise, if the check fails, it decrypts another string and repeats the procedure.

4.6.5 Native Behavior

Finally, the native libraries are executed (steps **#6A**, **#6B**, **#6C** of our methodology presented in Section 4.3). The goal is to investigate how malware can abuse legitimate actions a shared library can perform. To measure this aspect, *ANDani* processes the IPCFG of the native code, focusing on the analysis of the edges from native to Java and native to native (e.g., call to functions exported by other shared libraries).

Native reflection. A shared library might leverage the JNI callbacks to communicate with the Java world. Our measurement revealed that malware samples, especially the most recent ones, adopt this behavior more often than goodware.

Over the years, the usage of native reflection by malicious apps significantly increased, reaching over 90% in 2021. On the other hand, adopting JNI callbacks in benign software is present in 57.1% of the apps that use native components. In particular, among the several JNI callbacks available, we noticed that the usage of the `FindClass` and `GetMethodID` callbacks are about 35% higher in malicious apps than in goodware. In addition, *ANDani* succeeded in recovering over 81% of the arguments for goodware for both the callbacks. At the same time, it resolved less than 68% of `FindClass` and 70% of `GetMethodID` for malware from 2017. In half of the cases, both goodware and malware apps access classes and methods of the Android framework, while the remaining involve app-specific classes. About goodware, over 72% of these custom classes are present in the APK, whereas, in malware, this happens only in 21%. This second result could indicate how malware communicates natively with

Java components, not plain within the APK. These classes are in obfuscated files, retrieved online, and loaded at runtime.

We investigated further the classes and methods of the Android framework accessed with the native reflection, and we noticed a significant difference between the apps of the two datasets. In order of frequency, the native reflection is used by malicious apps to load a DEX or a JAR file through the `DexClassLoader` (or its superclass `ClassLoader`) class, get a handle to a system-level service such as with the `getSystemService` method of the `Context` class, interact with Android managers, inspecting incoming exceptions, and perform crypto and encoding operations. Adopting such techniques is approximately six times more frequent in malware than goodware, which is less than 4%.

Concerning the analysis of the Android managers, our result shows that malware mainly interacts with `PackageManager` to retrieve app information or verify the permission through the `checkPermission` method, `WifiManager` to check the connection, and `TelephonyManager` to retrieve sensitive information, such as getting the IMEI and IMSI with `getDeviceId` and `getSubscriberId` methods. Collecting unique identifiers for the device (IMEI) and SIM card (IMSI) is a well-known procedure malware uses to profile the victim or implement anti-emulator evasive techniques. However, these techniques are typical of the Java world, and malware samples moved this logic into the native layer over the years.

Moving to exploit the `ClassLoader` from native code, we noticed how recent malware loads the target Java classes by directly invoking Java methods, such as `loadClass`. This technique can replace the `FindClass` JNI callback, making the analysis much more complicated. Besides, we notice that malware in 2021 tends to leverage the Java reflection technique in the native code; for instance, the `ToReflectedMethod` and `FromReflectedMethod` callbacks are used four times more in malware than goodware.

Moreover, it is interesting to highlight how the malicious apps natively retrieve and handle the stack trace, using the `getStackTrace` method of the `Throwable` class to inspect its content. This technique is applied as a form of anti-hooking: by looking at the content of the stack trace, an app can detect the presence of either the Cydia Substrate or the Xposed framework (e.g., sample `4a7e`), as both manipulate the call stack [28].

Finally, whenever we found a native reflection pattern, we applied the technique described by Aafer et al. [2] to check if the invoked Java method needed no, normal, or dangerous permission. If the argument of `FindClass` could not be statically retrieved, we used the argument of `GetMethodID`. The Android framework method names are often unique, and in case we found multiple matches (like `read` or `open`), we excluded these cases from the analysis. First, more than 10% of malware, regardless of the year but with a gradual increase

in such a percentage over the years (up to 16% in 2021), invokes methods that require normal/dangerous permission; on the other hand, this percentage is negligible for goodware. Then, on average, comparing recent malware and goodware, we obtained respectively: 84% (vs. 94% for goodware) of the methods required no permissions, 8% (vs. 3%) required normal permissions, and 6% (vs. 3%) required dangerous permissions. This shows how malware samples abuse native reflection to perform privileged operations and, in particular, invoke methods for reading SMS, accessing the location, and reading contacts.

Library function. An ELF file might rely on external functions exposed by other shared libraries, in which symbols are dynamically resolved or included in the ELF file during the compilation (i.e., statically linked ELF files). Our analysis reveals that almost all the apps in the goodware and malware datasets import `libc.so` (which in Android include also `libpthread.so` and `librt.so`), `libm.so`, and `liblog.so` libraries. The only noteworthy relevant difference is related to `libz.so` (a compression/decompression library), used by 90% of malware and 6% of goodware. This discrepancy is because malicious apps often decompress components that will be used at runtime (e.g., sample `05b4`).

We identified several discrepancies between goodware and malware in the prevalence of usage of security-relevant functions (the ones summarized in Table 4.1). For instance, in 2011, only 3% of the malicious apps used `chmod`, and 7% used `mprotect`, which are respectively used to change the owner of a file and the permission of a mapped area in the memory. The use of these functions has grown steadily over the years to the point where, nowadays, more than 59% of the malicious samples in our dataset use the first function, and 87% use the second. On the other hand, these percentages are less than 15% for goodware

Moreover, the results highlight a sharp increase (more than half of the malware since 2019) in the usage of `inotify*` functions in recent malware; thus, we manually investigated their usage, identifying two common use cases: an anti-debugging technique and a trick to execute code when the app is uninstalled. For instance, `fcd4` monitors the files in the `/proc/<pid>` folder, such as `mem` and `pagemap` to check in real time a change in the memory mapping of the app' process and detect the presence of a debugger. On the other hand, `5264` monitors an empty custom file inside its installation folder to identify when its app is uninstalled and, as a result, opens a web page.

Then, we investigated the files opened through the `*open` family. In the first case, malware uses native code to open or access the files under the `/dev` and `/proc` folders more often than goodware. For instance, in 2021, 54% of malware and 2% of goodware opened the `/proc/version`. Another common target in the `proc` folder is `/proc/self/maps`, which describes the virtual memory in a process, and it is used by 84% of malware and only by

the 10% of goodware. Checking the contents of the `maps` file by an application can provide information about injected libraries, and it is a known technique used especially by malware to identify frameworks such as Frida (refer to Section 2.2). On the other hand, identifying access to device drivers located under `/dev` is another crucial aspect over the years; numerous vulnerabilities have affected that subsystem (e.g., the recent Use-After-Free vulnerability in the Android Binder [83]). The results highlight how recent malware is more prone to open drivers, such as `/dev/tty` to read the output of processes or `/dev/ashmem` to share large quantities of memory among processes, in which vulnerabilities have been found over the years [81].

Furthermore, some malware checks device-related information or open system-shared libraries, while the number of goodware is negligible ($< 0.1\%$). For instance, in 2021, 10% of malicious apps verify if an SELinux policy is enabled accessing the `/sys/fs/selinux/enforce` file, and 9% of malware explicitly interacts with the `/system/bin/linker` to load and run a dynamic executable, even if they are contained in a ZIP file. These new techniques have grown in the last years, performed by malware to detect the environment where they are executed or evade anti-malware controls.

Finally, for what concerns the analysis of the program executed through the `exec*` functions, it reveals that malicious apps leverage command line programs two times more than goodware to collect information related to or manage the running processes (e.g., `ps`, `pause`) and logs (e.g., `logcat`), or running shell scripts through a new bash (e.g., `sh`).

Dynamic loading. Dynamic loading refers to the ability to load and invoke functions of other shared objects at runtime without the need to link the library to the executable. In particular, this technique is based on two specific library functions: `dlopen` to load the library and `dlsym` to retrieve a pointer to the target function. The prevalence of this technique has increased over the years, from 2011, when 24% of malware employed it, until today, when the percentage is over 90%. In comparison, around 55% of benign apps in our dataset perform dynamic loading.

Looking at libraries and functions invoked with such technique, we found that for goodware, half of the loaded shared objects are well-known Android libraries, while more than three-quarters are for malware. Among the remaining quarter, most of the libraries are not included in the APK – we suppose these libraries are present in obfuscated files or retrieved from the internet and loaded at runtime (a manual analysis of some random samples confirms this). The most common libraries that are dynamically loaded are related to (de)compression and decryption/encryption operations; for example, the `uncompress` function of the `libz.so` library is dynamically loaded from over 90% of malware but only

from 0.4% of goodware apps. This finding again reinforces our idea that malware uses native components to prepare resources used at runtime to evade static analysis.

Even if some Android libraries are widely loaded from both goodware and malware, on average, their usage is very different, and this can be used to pinpoint suspicious operations. For instance, the `libc` library is loaded though `dlopen` by more than 30% of goodware and malware. However, malicious apps rely more on dynamic loading to invoke `libc` functions such as `__system_property_get` to retrieve the value of device-related properties and `chown` to modify the owner of some resource. In addition, we found an unconventional and rather peculiar use of these functions by malicious applications in which a few apps called `dlopen` and `dlsym` to obtain a function pointer to `dlopen` and `dlsym` themselves, and use that later on in the execution. Lastly, we observed how at least 9% of the recent malicious apps load Dalvik and ART runtime libraries, namely `libdvm.so` and `libart.so`, but this phenomenon occurs in less than 2% of goodware. It is crucial because such libraries can be (ab)used to bypass Android Runtime restrictions [220].

4.6.6 Main Takeaways

This section summarizes the main takeaways of our measurement of native code usage in Android apps, answering the first four research questions.

RQ1 & RQ2. Even before executing specific exploits, the primary purpose of native code in malicious samples is to implement evasive techniques and exploit its complexity to avoid the detection of analysis tools. For instance, investigating the `JNI_OnLoad` function is crucial in analyzing Android malware. In particular, using `RegisterNatives` with different hidden arguments, based on environment checks, to map different functions at runtime can be considered an anti-analysis technique for static and dynamic analysis – that, to the best of our knowledge, has not yet been documented.

In addition, malware exploits JNI and the native code to jump between Java and the native layers, making analyzing and identifying harmful patterns more challenging. Malware also leverages native code to protect itself from reverse engineering. For instance, the inspection of the native reflection confirms and brings to light new techniques for malware executing Java dynamic code loading, exploiting the native layer. Malware, therefore, tends to move malicious techniques from Java to the native layer (such as anti-hooking or accessing sensitive user information), and they start to replace JNI callbacks with a direct invocation of Java methods from the native layer.

RQ3. The results highlight how malware increasingly abused native code over the years as a protection mechanism against static and dynamic analysis tools by implementing evasive controls and harmful operations into the native layer. Also, they are moving Java layer-specific operations in the native code.

Malicious samples have adapted to support more and more architectures. This can indicate many factors, the first of which is how today malware samples exploit native code for their malicious purposes, trying to be effective on as many devices as possible. Using the same name for different libraries is a clear sign that the dispatch (Step 2 of Figure 4.1) might change depending on some checks, and changing the file's extension is a trivial and old trick, but nowadays, it is still common.

Finally, we have noticed that the reuse of malicious native libraries is a frequent practice that has lasted over the years, showing carelessness in concealing it.

RQ4. The analysis reveals that goodware is much more adherent to good practices, while malware tends to exploit native code for security-relevant actions, indicating operations that require further investigation.

First, the native functions declared by benign apps are reachable in most cases from the main apps' code, and the trigger for their execution is very often dependent on user interaction. Conversely, malware exposes a significantly different trend: the average number of reachable native functions from the main app codes has decreased dramatically, consequently implying a solid presence of DCL. Moreover, we noticed how the invocation of native methods is almost immediate and occurs mainly without user interaction.

Second, goodware is much more prone to load libraries via the `loadLibrary` method, thus making the analysis easier – given that they are loaded from a single location and must be present in the APK. This was the same trend observed in malware until 2015, while it is changing in favor of using the `load` method. This fact brings numerous problems to static analyses as it may not be possible to know what library will be loaded or where it is located in advance.

Regarding the operations done by native code, malware is more prone to call security-relevant library functions. We highlighted the high discrepancy in using `libz` for (de)-compression and network-related functions retrieving resources at runtime. In addition, malware often reads the content of some particular files to perform environment controls (e.g., emulator verification), interacts with low-level components (e.g., linker) to bypass standard checks, and runs command line programs more frequently than benign samples. Most malware also abuses the dynamic loading of DLLs (i.e., `dlopen` and `dlsym`), and, very often, the loaded library is "generated" at runtime, while goodware does the opposite. This

reinforces the consideration that malware is more prone to prepare resources (e.g., decrypt or download code) from native code in an evasive way. Moreover, most loaded functions could hide suspicious operations, such as (de)compression or permission management.

Finally, the analysis of the native layer shows how malware and goodware have native components in common to protect their code from reverse engineering (e.g., protector or packer), whose use requires a more in-depth analysis to shed light on this grey area.

4.7 Use case: binary classification

The previous section discussed the many facets of native code execution in Android apps and highlighted core differences between how benign and malicious apps use native code. In this section, we test to what extent the suspicious tags assigned by our system can be used and whether they are useful for malware detection tasks. Moreover, this section verifies which malware characteristic is more relevant to detection. For this task, we built a dataset using all the 15,647 goodware at our disposal and sub-sampling the same amount of malicious apps collected in 2021. We selected all samples classified as "Singleton" (15,427) (those for which AVClass2 could not determine the family), and we sampled the remaining 220 malicious apps one per family to avoid bias towards particular families. We extracted the suspicious tags defined in our methodology for each sample and created a vector of 74 features (62 booleans and 12 floats). Table B.1 in the Appendix reports each tag with the data type used in the vector.

In detail, we used a Random Forest classifier because it handles numeric and categorical features without needing encoding. We set the split criterion of the algorithm to be the Gini impurity and tune the remaining hyperparameters (such as the number of trees, their depth, and the number of features to consider when splitting a node) by using the Out Of Bag (OOB) error computed during the training phase. As reported in Figures 4.2 and 4.3, the optimal OOB error is obtained when considering 141 trees with depth 32 and the *sqrt* as a metric to define the number of features to test when extending a node. The optimal value for the number (of trees for Figures 4.2 and depth for Figure 4.3) has been chosen using the Elbow method after calculating when there is no performance improvement.

We used a 10-fold cross-validation approach to train and test our classifier, each round training the model on k-1 folds and testing its performance on the remaining fold – different for each round – to measure how well the classifier generalizes on unseen samples and limit the bias introduced by the model. Table 4.4 summarizes the average performance obtained on the test sets, including accuracy, precision, recall, and F1-score. Even though we opted

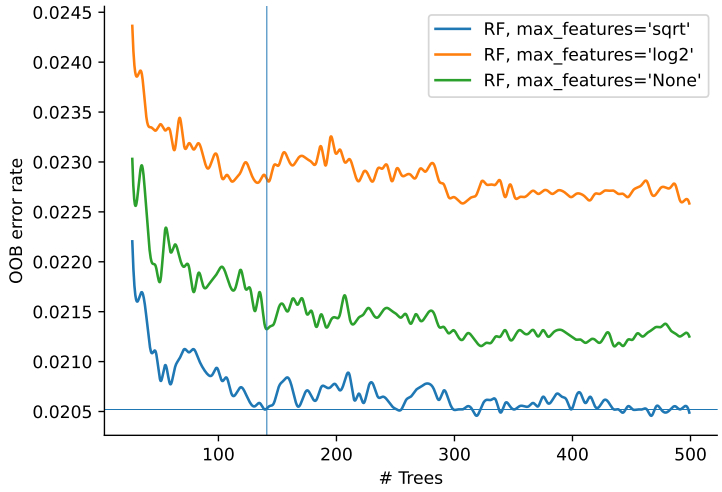


Figure 4.2 OOB error defining the optimal number of trees and features.

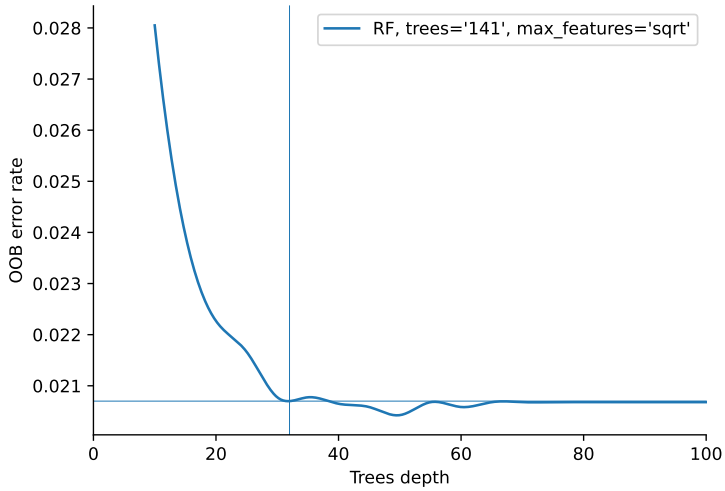
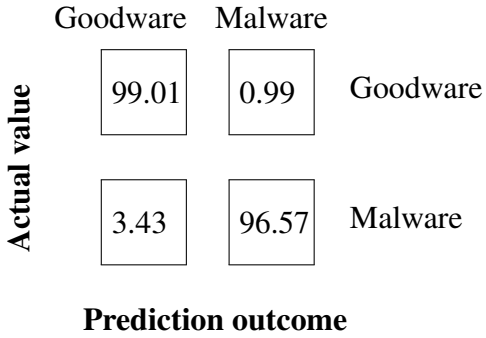


Figure 4.3 OOB error defining the optimal depth of trees.

for a simple classification scheme, our results were surprising. In fact, by simply leveraging the suspicious tags to distinguish between goodware and malware, the average error rate was 2.21%, with an accuracy, respectively, of 0.99 and 0.96 and a mean F1-score of 0.97.

Feature importance. We ranked the features used in the classifier based on their Mean Decrease Impurity (MDI). We reported in Table 4.5 the top 10 of them together with their relative importance normalized as a percentage.

Table 4.4 Left: confusion matrix; right: classification report



Metric	Goodware	Malware
Precision	96.57	99.02
Recall	99.01	96.57
F1-score	97.77	97.78
Accuracy	99.01	96.57
Accuracy	97.79	

Table 4.5 Top 10 features sorted by MDI score

Tag Category	Tag Title	MDI score (%)
J_NATIVE_METHODS	NO_REACHABLE	15.88 %
J_LOAD_METHODS	APP_LIFECYCLE_EP	13.17 %
J_NATIVE_METHODS	APP_LIFECYCLE_EP	11.50 %
J_LOAD_METHODS	PATH_LOAD_METHOD	8.87 %
STRING	PROPERTIES	7.15 %
J_NATIVE_METHODS	ACTIVITY_LIFECYCLE_EP	5.90 %
SUSP_LIB_CALL	MEMORY_PROTECTION	4.02 %
SUSP_LIB_CALL	PROCESS_MANAGEMENT	3.65 %
SUSP_LIB_CALL	IDENTITY	3.45 %
SUSP_LIB_CALL	PERMISSION	2.83 %
	Average	1.35 %
	Standard deviation	0.24 %

The tags belonging to the categories `J_NATIVE_METHODS` and `J_LOAD_METHODS` (the first two steps of our methodology, which capture native/load methods in the Java code and the entry point from which they can be reached) accounts for almost 50% of the total feature relevance. This shows that the insights gleaned in Section 4.6.1 and 4.6.2 represent the most discriminating traits for the classification of goodware and malware; namely, our results suggest that a reliable indicator of malicious behavior is when an app reaches the native code without user interaction and such native code is not statically available. Although we observe higher importance for the first two features, the average relevance suggests that each part contributes to the classification task, reflecting the importance of combining suspicious tags to identify malware and goodware.

Furthermore, four of the top-10 entries belong to the category `SUS_LIB_CALL`, which denotes the use of security-relevant functions within the native code. Interestingly, the most impactful is the 'Memory Protection' calls `mmap` and `mprotect`, a prerequisite to executing

dynamically loaded code. Finally, the fifth entry indicates the presence of `build.prop` key strings. The `build.prop` is a file that contains build properties and settings in the format `key=value`. Some contents are specific to the device or manufacturer, while others vary according to the operating system version. Retrieving these values significantly impacts security because attackers can fingerprint the device and engage in evasive behavior or select a valid exploit for the system.

Classification errors. In the last part of our analysis, we investigated the root causes of classification errors and whether those were attributable to any particular characteristics of the samples. In our setting, we define a false positive (FP) as a classification error in which a benign sample is labeled as malware; vice versa, the classifier produces a false negative (FN) when predicting malware as goodware. We repeated our classification task 100 times by using independent folds for each experiment, thus resulting in training and testing 1K different classifiers. The rationale behind this choice is to isolate those samples that are always mispredicted as FPs (0.5% - 81/15,647) or FNs (3.1% - 483/15,647). These samples were further investigated by analyzing the tags extracted with our methodology and resorting to manual reverse engineering for a subset.

When narrowing down to FNs, we detected that the model errs when the malicious logic (e.g., sample `c227`) is fully contained in `classes.dex` files – which is out of the scope of this work, and it includes well-known legitimate native libraries. For example, sample `58b3` only ships two open-source libraries in the standard location, namely LAME, to manipulate MP3 files and a second one that provides WebRTC capabilities. However, many samples contain the definition of some native methods in the Java code but do not contain the relative ELF file in standard or non-standard locations. The respective sample is then characterized by the sole presence of the `NO_ELF_NAME` tag (`J_LOAD_METHODS` category), whereas almost all the other tags are missing. In such a case, the model needs more information for an accurate classification.

On the other hand, we discovered that misclassifications of goodware (FPs) are mainly due to the heavy usage of Dynamic Code loading (`DYNAMIC_LOADING` category) or suspicious library calls (Table 4.1). In particular, almost all FPs are samples with native libraries that try to protect the intellectual properties of the developers with integrity checks, obfuscation, or packing techniques. By nature, such techniques generate exactly surreptitious code that represents the core of our analysis.

RQ5. The understanding of the native layer is a crucial aspect in the cat-and-mouse game between malicious and benign actors for the detection of Android malware. The most noteworthy result is that the significant differences in how malware and goodware samples

interact with the native code heavily affect the classification task, even more than their operations. Moreover, the most important native features are related to how the native libraries kick into the execution of an Android app.

4.8 Related work

There are mainly two areas of work relevant to this chapter: the analyses focusing on the Java layer and those considering JNI. This section discusses both and highlights the differences between our methodology and *ANDani*.

Java layer analysis. In 2013, Octeau et al. [158] implemented Epicc, an analysis framework based on Soot – a Java optimization framework proposed by Vallee et al. [226] – to resolve Inter-Component Communication (ICC) in Android apps. In 2014, Arzt et al. [22] proposed FlowDroid, a dataflow analysis framework for taint detection of the Java code of an Android app. A context, object, and flow-sensitive taint analysis considers the Android app lifecycle. FlowDroid extends the Soot framework and creates an app-level dummy Main class to collect all Android system events. In the same year, Wei et al. [234] proposed Amandroid to conduct static analysis for security vetting of Android apps. It builds a context and flow-sensitive inter-procedural control flow graph (ICFG) of the whole app and computes the point-to information to detect several security-related problems. In 2015, Li et al. [131] proposed a new static taint analyzer to detect privacy leaks among components in Android apps, named IccTA. It propagates the context information among different components to resolve call parameters and return values. In the same year, Gordon et al. [107] proposed DroidSafe, a static analysis framework able to resolve ICC and Remote Procedure Call calls to detect potential data leaks by tracking information flows. Then, Yang et al. [244] proposed AppContext, a static analysis approach to extract context security-sensitive behavior to assist the app analysis focusing only on the Java layer. In 2021, Wu et al. [237] proposed BackDroid, an inter-procedural analysis of Android app, with the primary goal of improving the performance of the static analyzer described earlier by implementing a novel technique named *on-the-fly bytecode search* which searches the disassembled app bytecode text just in time when a caller needs to be located.

JNI analysis. In 2012, Yan et al. [243] proposed DroidScope, an emulation-based Android malware taint-analysis engine used to analyze the Java and native components (x86 and ARM architectures) of an Android app to track information leakage. In 2014, Qian et al. [174] performed the first large-scale study on information flows using JNI. This study leverages

NDroid, a novel dynamic taint propagation tool based on QEMU, which tracks JNI and system library functions in Java and native code. Alfonso et al. in 2016 [3] performed an extensive analysis on the adoption of the native code on Android apps, highlighting potential usage of JNI, and proposed a new method to generate a native code sandboxing policy automatically. The same year, Sun et al. [209] proposed TaintART. This customized ART compiler inserts the taint logic and retains the original ahead-of-time optimizations that perform taint analysis to track data flow. Rasthofer et al. [176] proposed Harvester, a hybrid analysis tool that combines static backward slicing to identify interesting code with the execution of the code for extracting runtime values. In 2017, Alam et al. [8] proposed DroidNative, a static Android malware detector based on the analysis of the native code. It introduces the Malware Analysis Intermediate Language (MAIL) concept to create a high-level representation of the native code, which is then used to build a behavioral signatures template.

Xue et al. [242] presented Malton, a dynamic analysis platform built on Valgrind for malware detection based on information flow tracking on Java and JNI code. In 2018, Wei et al. [236] presented JN-SAF to conduct static cross-language dataflow analysis of Android apps to track information leaks through the Java and the native parts. JN-SAF builts the analysis of the Java part of the app on top of Amandroid [234]: the study of the native components instead – for both 32 and 64-bit versions of ARM, MIPS, PPC, and Intel architectures – relies on the Angr's symbolic execution engine [230], In 2019, Lee Sungho [130] proposed a novel JNI program analysis technique that combines the analysis of Java and C code separately to extract semantic summaries of C code from JNI programs. In 2020, Andarzian et al. [13] proposed the CTAN framework, which extends JN-SAF to improve its performance. The same year, Fourtounis et al. [70] proposed an approach to recover JNI callbacks in the native code: disassemble native binaries, recover static symbol information, and produce a model for statically linking the native callbacks. In 2021, Samhi et al. [188] proposed JuCify, a framework that combines Android bytecode and native code into a unified model to detect data leaks. The native code analysis is built on top of Angr, while the Java code analysis and the unified model rely on the Soot framework.

4.8.1 Comparision with state-of-the-art tools

During the design phase of *ANDani*, we investigated the state-of-the-art tools for static analysis to understand which technologies are best to rely on.

Most works focus on the Java layer and do not handle native code. However, we noticed that there is a tool in common among the most cited works (e.g., Epicc [158], and Flowdroid [22]): the Soot framework [226], which is still under active development [108], and therefore it was our choice since the alternatives are no longer supported (e.g., Amandroid [234]). However, as already discussed, we had to improve the analysis of some typical Android mechanics, namely, the detection of entry points and concurrent execution management.

Then, for inter-language analysis, JN-SAF [236] and JuCify [188] are the most recent ones; however, these tools perform taint analysis between different code layers to detect data leaks. As for the analysis of the Java layer, the former is based on Amandroid, while the latter is based on Soot. On the other hand, they both use Angr [230] to analyze the native code.

We tested Angr with the same configuration of JuCify. However, from several preliminary tests on native Android libraries taken from real-world goodware apps (the same we used in Section 4.5), we discarded it because the one-hour timeout was often reached (because of the path explosion typical of symbolic execution), thus making it unsuitable for our large-scale measurement. Moreover, JuCify's approach is not suitable for our needs. Using Soot, they lift Java code to Jimple, a three-address intermediate representation provided by Soot; then, they extended Angr to lift native code to Jimple so they could reason on a unified model. The issue is that we are interested in every single instruction in the native code, as we need as much precision as possible; instead, JuCify considers call instructions because its purpose is to create a call graph while we need the interprocedural control flow graph. Regarding JN-SAF, it is no longer maintained and does not investigate specific aspects of native code we are interested in (e.g., library calls and their arguments). Given that it is also based on Angr, we did not investigate further and then decided to write the analysis of native code from scratch.

4.9 Discussion

This work significantly differs and complements all state-of-the-art tools that deal with native code for the type of analysis performed and the study's goal. In particular, *ANDani* is not limited only to a Java to native dataflow analysis: it also analyzes all aspects of the native components, from the entry point analysis and the triggering condition of JNI methods to the suspicious library function invoked by the native code. This design can improve or make

the automatic detection of Android malware in a binary classification task more robust by including the native-related features extracted by *ANDani*.

However, the *ANDani* implementation has all the intrinsic limits of the static analysis, particularly the backward taint data analysis [132]. In addition, if the code that is dynamically loaded is not present in the APK, we are not able to analyze it, and given the current spread of droppers in the PlayStore [221], it is an issue that will need to be addressed.

Chapter 5

Android, Notify Me When It Is Time To Go Phishing

5.1 Introduction

The complexity of the native layer in Android apps is heavily abused by malicious authors and benign actors as a protection mechanism to make analyzing their apps more challenging. Also, the native layer exposes powerful features that several malware samples exploited (and continue to exploit) to attack both the Android operating system and other apps. Of all available services, the *inotify* [136] APIs allow a userspace program to monitor and react to file system events (e.g., file accesses, deletions, and modifications), targeting either singular files or entire directories and filtering only specific events. Starting from version 2.6.13, the Linux kernel includes inotify, and Android has fully supported it since the early versions (API Level 1 [92]). In such a scenario, apps can *watch* file system *events* – employing the inotify terminology – from the native code through the standard inotify APIs, and the Java layer, using the `FileObserver` class.

While providing powerful primitives to implement new features, the inotify infrastructure has also played a role in Man-in-the-Disk (MitD) attacks [138]. These attacks are possible when the program's logic depends on the content of a file that an attacker can access and modify. Inotify proved perfect for this task. By watching for a specific event on the target file, the attacker can timely and opportunistically replace its legitimate content with malicious data. The consequences may be more or less severe depending on the tampered data.

For example, in 2018, Google security researchers revealed that the popular game Fortnite for Android was vulnerable to such attacks [31]. In particular, the app distributed on the app stores did not contain the game code but was downloaded from the publisher's website.

During the download phase, the app stores the partial data in the device's shared storage, moving it to its private directory once the download is completed. By relying on inotify, an attacker could seize the right moment and tamper with the partial game code file to inject a malicious payload, effectively fooling Fortnite into installing a fake app instead of the intended one. In this way, an attacker can gain the code execution privilege in the context of the original app.

In 2016, Ahn et al. [6] showed that particular file system events' occurrences allow the detection of a specific UI screen transition. Once again, inotify made it possible to monitor specific file system patterns, this time for perfectly timing a phishing attack by starting an activity that resembles the original one. While inspiring, this work relies on capabilities that attackers can no longer obtain under reasonable threat models (e.g., Android does not allow third-party apps to list running apps and their PIDs).

Given its history of misuse for malicious purposes, an aspect has been understudied. This work assesses whether the inotify subsystem can still play a role in *state inference attacks* in modern Android malware despite the tight security models the Android developers put in place. This attack class aims to determine the state of another app and pave the way to a large class of attacks, including *phishing*. In particular, this work focuses on determining when an app starts, which, at the same time, is the most valuable for an attacker and the most practical to analyze automatically. From an attacker's perspective, determining when the user willingly launches an app maximizes the chances to lure him into providing account credentials to a phony UI interface. It is not by chance that banking malware often monitors apps' startup for phishing purposes [232, 126]. By choosing to study the app's startup and not any generic UI transitions, we do not need heavy human intervention to select and trigger valuable ones (e.g., login screens). The concept behind our approach is straightforward: if an app always generates a *unique* file system event at startup (among other apps installed on the device) and an attacker can monitor such an event with inotify, she can easily infer when the user has just opened the app.

First, we manually tested the feasibility of such an attack by creating a vulnerable target app that reads a world-readable text file when its main activity starts and a malicious app that monitors these events with inotify. When the vulnerable app was executed, the malicious app was "notified"; thus, it could perform a phishing attack by pushing a new "phishing" activity if the attacker has enough permissions. Picture the scene: the user taps on the target app icon, and the new fake spoofed activity that looks like the original one is displayed on the screen; this leaves no trace to the victim, who has no way of noticing the deception and distinguishing between the original and the malicious apps.

However, this precise attack is limited by a privileged Android permission (`SYSTEM_ALERT_WINDOW`) that requires a slightly more complicated procedure to get it from the victim. Although this does not stop modern malware authors from using it (according to our measurements, 57% of malware requires it, while this percentage plummets to 9% for benign apps), the attack can still be performed by popping up a fake notification that seems to come from the target app.

For this reason, this chapter investigates the conditions surrounding the feasibility of this attack, trying to answer the following research questions:

RQ6: Are malware authors already exploiting inotify?

RQ7: What capabilities (e.g., permissions or information) must an attacker have to carry out this attack?

RQ8: To what extent does the Android version affect this attack and the attacker's capabilities?

RQ9: Are the attacker's capabilities realistic to carry out this state inference attack and mount a phishing attack?

RQ10: How is the security posture of other vendors concerning this attack?

RQ11: Which mitigations can stop this attack for good?

First, due to Section 4.6 showed that malicious samples natively abuse inotify to react to some specific events, we investigated the usage of inotify among modern Android malware samples to assess the novelty of our attack. Our results on over $10,000$ of malware in 2021 show that they abuse inotify only for implementing anti-debugging tricks or executing code when they are uninstalled. In the other cases, we detected inotify just from analytics libraries. Therefore, no existing Android malware implements the presented attack pattern.

Then, we investigated the conditions for carrying out this attack. Given that the attacker's capabilities depend on which files the apps use when started and if she has permission to add an inotify watch on such files, we developed *inoTool*, a tool to dynamically analyze Android apps and monitor their interactions with the file system.

With *inoTool*, we analyzed a heterogeneous dataset of $4,863$ benign Android app, created by collecting the most downloaded free apps over the 50 categories in the Google Play Store. We found that an attacker can generate a kind of "signature" that uniquely identifies which files her target app opens at startup, enabling her to launch a phishing attack by monitoring one of such files. This attack is feasible on a large scale, and during our investigation, we uncovered some design issues that improve the likelihood of success. For example, we discovered an information disclosure vulnerability through which one can get a partial list

of the apps installed on a device, despite the recent changes in Android strongly restricting package visibility, namely, querying for information about the other apps.

We identified three attack models by analyzing the results and the mechanisms involved. As the main finding of our work, in the most potent model (in which an attacker is assumed to know the installation path of its target app), ***all*** Android apps are vulnerable, regardless of the Android version (till now). Moreover, the only capability needed to implement this attack is protected by a single, normal permission automatically granted at installation time. Although Google is aware of the risks deriving from this permission and performs capillary checks on its Play Store, we will also show that the current Android security model presents several loopholes that make obtaining this permission unnecessary.

The second attack model assumes that an attacker can no longer obtain the installation paths of the apps installed on the device. Under these circumstances, we show that an attacker still manages to implement the attack on a limited number of apps. The third and last model refines the strategy further and manages, on average, to target half of the apps installed on a typical device.

Finally, this chapter shows how different remedies defeat these types of attackers. However, in some cases, the specific behavior of the target app, in combination with the apps installed on the device, still leaves a small window for successfully carrying out the attack.

To sum up, this chapter provides the following contributions:

- a novel methodology to perform phishing attacks on the Android ecosystem based on file system-based signatures that uniquely identify the startup of every app;
- *inoTool*, an analysis tool for Android apps that compute their signatures based on file system events. In the spirit of open science, we released the source code we developed during this study [182];
- a deep analysis of the likelihood of a successful attack concerning the capabilities of an attacker and the version of the Android operating system in which the malicious app is running.

5.2 Background

This section provides the additional technical background for the rest of the chapter. It starts by presenting the relevant details of the Android operating system, and then it continues by introducing the *inotify* subsystem of the Linux kernel.

Mandatory Access Control & SELinux. Security Enhanced Linux (SELinux) is a mandatory access control (MAC) system for the Linux operating system, and it was introduced in Android 4.3. A MAC system differs from Linux's discretionary access control (DAC) system. Indeed, with a DAC system, an owner of a particular resource controls access permissions associated with it, while on a MAC, there is a central authority for a decision on all access attempts. In Android, DAC and MAC coexist: a user can interact with a resource if she has enough permissions (DAC) and the security policy allows it (MAC).

In detail, SELinux adds opaque pointers to potentially sensitive kernel objects (e.g., files, network interfaces) and mediates access to such objects by placing *hooks* functions to determine if an action (known as *permission*, such as `read` or `write`) should be allowed. The decision is made according to the *policies*, which are a set of *rules* that guide SELinux in the choice based on the *labels* of the resources. Every object (e.g., files, directories, processes, etc.) has associated a *security context* (or *label*) that specifies its security-related information, i.e., SELinux user, role, type, and sensitivity. In particular, Android relies on the Type Enforcement (TE) component of SELinux, in which the policies are enforced based on the type of subjects and objects. By default, SELinux denies all requests except the ones that respect the current policy. In addition, on Android, a third-party app is assigned to the context with the type `untrusted_app`.

A SELinux policy rule comes in the form `allow source target:class permissions`. This rule allows a subject (of type *source*) to access the object (of type *target*). The *class* field specifies the kind of the object (e.g., file, socket, etc.), while the set of involved operations is specified by the *permissions* field. For instance, the `allow untrusted_app app_data_file:file {read write}` policy specifies that untrusted apps are allowed to read and write files labeled "app_data_file". Finally, Android provides a set of macros to handle the most common cases to simplify the rule definition. For instance, it defines the permission `rw_file_perms`, which automatically includes all the operations needed to read and write a file.

Android external memory. Handling external memories (especially SD cards) in Android has changed a lot between versions. In the past, `/sdcard/` folder was used for this purpose, while today it is a symbolic link to the `/storage/self/primary/` folder, which in turn is always (counterintuitively) a symbolic link to `/storage/emulated/0/`. However, regardless of the presence or absence of a physical external mass memory device (i.e., SD card), the Android system always "emulates" an external device under the folder `/storage/emulated/`, where it mounts a partition of its internal memory.

When a user inserts an SD card, the system offers her two configurations. The first, named *portable*, manages the memory space on the SD card so the user can remove and use it on other devices. When a new SD card is plugged in, the system creates the mount point `/storage/<sdcard_ID>/`. Instead, the *internal* configuration extends the device's internal memory. Under the hood, the system mounts the SD card in the emulated external storage, i.e., under `/storage/emulated/`. The SD card cannot be transferred between different devices in this scenario.

Inotify in Linux and Android. Many user apps in modern computer systems need access to the file system beyond simply reading/writing files. Take a remote directory synchronization utility as an example. This utility must monitor the target directory to detect file creations, deletions, and modifications to operate automatically. While possible, implementing this type of monitoring by employing traditional file system operations (e.g., listing the directory every second by relying on a busy-waiting model) is not efficient. For this reason, modern operating systems provide dedicated APIs for implementing low-overhead file system monitoring.

One such API is the *inotify* subsystem of the Linux kernel, which provides an event-driven interface to intercept file system events. The canonical inotify workflow starts with a program creating an inotify instance through the `inotify_init` syscall and specifying what type of events to monitor through the `inotify_add_watch` syscall. The inotify instance can then be queried with the `read` syscall that blocks the program execution until one of the monitored events occurs.

The Android operating system wraps this workflow in the `FileObserver` abstract class, which programmers can use as a base class to implement file system monitoring. This class constructor takes the file paths to monitor as input, while two more methods start and stop the file system monitoring. Finally, the app can specify the event handling logic by implementing the `onEvent` abstract method that the framework invokes each time an event involving the monitored paths occurs.

5.3 Related Work

In 2011, Felt et al. [64] first discussed the risk of phishing attacks on mobile platforms and provided a set of techniques to carry out such attacks. As the authors pointed out, an essential ingredient of a successful phishing attack is to intercept the execution of a legitimate app promptly and to overtake the UI to present the user with a faithful copy of the interface of the intended app. When facing a seemingly familiar interface, the user is lured into providing sensitive information, such as credentials. While back in 2011, it

was relatively easy to determine when an app started (any app could list all the processes running on the system and implement a *poll-like* wait behavior), things became much harder once phishing attacks became prevalent. As a result, process listing was soon forbidden, forcing attackers to find new ways to detect when the user launched an app. These new techniques are commonly known in the literature as *State Inference Attacks*. Researchers showed that real-world malware [126, 154, 232] employ state inference as the first stage in their attacks, either for *phishing* or to monitor and profile the user. Several research works have studied and documented techniques to reliably determine when an Android app is being executed [39, 179, 65, 71, 203, 204, 16, 172]. Possemato et al. [172] grouped all existing vulnerabilities into two macro-categories:

I) File system layer. The `proc` file system (i.e., `procfs`) is a pseudo-file system that provides an interface to kernel data structures. The files in the `procfs` contain sensitive information, such as the program's name currently executed. Several works [30, 39, 203] discussed different techniques that allow a malicious user to build state inference attacks leveraging the `procfs` information. In addition, Spreitzer et al. [203] proposed a fully automated technique to assess the `procfs` information leaks.

II) Android System Services layer. Android apps use the System Services APIs of the Android framework to access Android-specific features or interact with hardware components. To interact with privileged components, Services interact with the `system_server`, which runs as the privileged system user. A vulnerability in some Service APIs can lead to the disclosure of sensitive information, opening the door for state inference techniques. Many works [65, 204, 172] investigated these aspects, showing many vulnerabilities of the Android components. For instance, Possemato et al. [172] detected several vulnerabilities, among which one that allows an app to collect statistics of other apps through the `getProcessMemoryInfo` API of the `ActivityManager` without any special permission.

To mitigate these attacks, Google has deployed several patches and countermeasures. First, Android strengthened the data returned from the APIs, removing other apps' sensitive information. For instance, the `getRunningTasks` and `getRunningServices` APIs of the `ActivityManager` class returned the list of the tasks and services currently running in a device. These methods have been deprecated from Android versions 5.0 and 8.0; nowadays, they return only the information of the caller app (and well-known non-sensitive data). In Android 7.0, access to `procfs` is limited to very few harmless files. As a result, to our knowledge, all documented file system-based state inference attacks no longer work on

modern versions of Android. However, despite these mitigations, some recent works have shown that phishing attacks are still feasible [16, 35].

On the defensive side, some works have been proposed over the years. [238, 145, 180] tried eradicating the phishing problem on Android, preventing a malicious user from spoofing the UI. The authors do not avoid a malicious app to perform the state inference attack on which the phishing is based but block or react to malicious UI that seems legitimate. Only "LeaveMeAlone" [248] tries to detect and block a malicious app that performs inference attacks (or tries to steal sensitive data) by analyzing shared resource usage. This tool leverages static information (e.g., permissions) and the `procfs` files to monitor the device apps, but it will not work on Android 7.0 and newer due to the enforcement of the `procfs`.

Finally, in 2016, Ahn et al. [6] proposed *inishing*, an inotify-based user interface (UI) phishing attack, which at the time of writing is the work most closely resembling ours. They show how malware can monitor *access patterns* (i.e., a sequence of file system events), enabling it to detect the transition from one UI screen to another. Once the access pattern events are triggered during a screen transition, the malicious app needs to check whether the target app generated it. The authors retrieved the PID of the target app through the `getRunningAppProcesses` API (assuming it is already running). They verified whether it is in the foreground by inspecting the files in the `procfs`. However, since Android 7, Google has significantly restricted access to the `procfs` folder, and only system apps can obtain the list of the running processes (and their pid). These restrictions make this attack infeasible, but as we will show, there is still room to abuse inotify.

5.4 The use of inotify among malware

To verify the novelty of the inotify-based state inference attack, this section investigates if and how current malware exploits inotify. To this aim, we collected and analyzed $10,000$ Android malware of 2021 from AndroZoo [12]. We considered malicious the samples with at least five anti-malware detections on VirusTotal.

For this type of study, it is crucial to distribute the malware families as uniformly as possible; therefore, we considered a maximum of 5% for each family. Given that Androzoo does not indicate the family of the malicious samples, we downloaded each sample's VirusTotal report and leveraged AVClass2 [192] to determine the corresponding family. Finally, AVClass2 classified 140 families and 8400 samples as *Singleton* – the term used to define the samples for which it could not determine the family. We report the percentages of the five most frequent families and the Singletons in the second row of Table 5.1.

Table 5.1 Distribution of the malware families used to verify the novelty of the inotify-based state inference attack

Dataset	# of Samples	# of Families	Fam1	Fam2	Fam3	Fam4	Fam5	Singleton
	10,000	140	5.0	0.8	0.8	0.7	0.5	84.0
Java layer	744	65	4.8	3.9	2.4	2.1	2.0	57.8
Native layer	769	30	0.23	0.19	0.12	0.09	0.08	98.7

As explained in Section 5.2, an app can install an inotify watch from both the Java and the native layers. To account for either case, we used two software components to analyze Android apps statically. The goal of these components is to resolve the arguments – performing a backward iter- and intra-procedural taint data analysis – of the API functions that interact with inotify to determine the path of the file that will be monitored.

Java layer. The first component is an extended version of the Soot [226] framework to analyze the DEX files and resolve the constructor's argument of the `FileObserver` objects that takes in input the path of the file to monitor. When the tool resolves a `java.io.File` object, it performs a backward taint analysis until it reaches an Android API method invocation (e.g., `getExternalFilesDir`) or a string. This procedure is repeated recursively if it finds another `File` constructor. The output of this component may be a string with the full file path or a combination of Android-specific APIs. For instance, if an app installs an inotify watch on the file named `example` in its folder on the external memory, and it uses the `getExternalFilesDir` method (of the `android.content.Context` class) to retrieve the path of the external memory, the output is the concatenation of the Android API and the file name: `android.content.Context.getExternalFilesDir()/example`.

The analysis reveals that 7.4% (744/10,000) of our malicious apps, distributed over 65 different families, instantiates `FileObserver` objects. Among the 744 samples, we measured 57.8% (430/744) Singleton, and we did not observe a predominant family (as reported in the third row of Table 5.1).

Most resolved paths depend on Android APIs to return the path of the "private" folders, where the app can place persistent files it owns. We also investigated which Android APIs are used to retrieve the path(s) of the private folder in the external memory. Table 5.2 reports all the "tainted" APIs we found in malware, that is, the ones involved in generating the path passed as input to the `FileObserver` class.

Given that in almost all these cases, the malware installs a watch on a file under these folders, there is no particular interest in monitoring files of other apps. Manually looking at folder and file names, we found that these cases are mainly analytics libraries. For instance, the

Table 5.2 Tainted Android APIs that affecting the `FileObserver` constructor in malware

Path	**%**
`android.content.Context.getFilesDir`	11.32
`android.content.Context.getNoBackupFilesDir`	7.84
`android.content.Context.getExternalFilesDir`	4.35
`android.content.Context.getDir`	1.74
`android.os.Environment.getExternalStorageDirectory`	0.44

Yandex [20] analytics library monitors the crashes of an app watching the file `appmetrica_-crashes` in the app's file directory (i.e., `android.content.Context.getFilesDir()/-appmetrica_crashes`). The most common target is the folder `/data/anr`, which was detected in 22.2% (165/744). When the UI thread of an Android app is blocked for too long, Android stores logs of the "App Not Responding" (ANR) errors in that folder.

The only pattern we found where the samples seem to monitor file system events generated by folders shared with other apps was present in the 4.2% (31/744) of the samples across eight families – disregarding the Singletons, the most prevalent (5/31) is the trojan `triada`. Such samples retrieve the path of its folder in the external memory and install an inotify watch for all the parent folders. However, this code belongs to the Vungle [229] library for advertising and monetization.

Finally, in some cases, the tool could not recover the path because I) the sample concatenates strings at runtime (26.7%), II) it is passed through an (external) Intent (0.3%), or III) it is read from the shared preferences (0.1%). Therefore, we manually reverse-engineered some samples, one for each family, and then moved to the Singletons. During this manual investigation, we did not find any new malicious pattern.

Native layer. As a second component, we leveraged *ANDani* to analyze the native libraries, identify the calls to the `inotify_add_watch` API, and resolve their inputs. In the malware dataset, 87.4% (8,742/10,000) of the samples contain native code, and the analysis revealed that 77.0% (6,731/8,742) of the malware samples use the inotify API in just 339 different native libraries. Given that we did not find a predominant family (fourth row of Table 5.1), this low number of libraries w.r.t. a high number of samples is because these samples share such libraries. Most analytics libraries, such as Umeng [245], monitor files in their private directory to identify app failures.

The analysis of native libraries proved to be more complicated than the Java one; our tool failed to resolve the argument to the `inotify_add_watch` function in more than 40% of cases because the string was not statically available. Therefore, we resorted again to manual reverse engineering, using the same strategy discussed above, stopping when we did not

notice new patterns emerge. With this mixed manual and automatic approach, we uncovered two "malicious" cases (already introduced in Section 4.6.5). First, an anti-debug strategy that we often (but not only) found implemented in the Jiagu packer. To give a practical example, we analyzed one sample (e.g., sample `9f02` – refer to Table A.1) that creates a new thread, and from this thread, it uses inotify to monitor the `mem` and `pagemap` files in the `/proc/<pid>` folder. In this way, it checks in real-time a change in the memory mapping of the app process to detect the presence of a debugger.

Second, we observed some apps that monitor if specific files inside their installation folder are deleted (e.g., the `base.apk`). This way, they can identify if the app is about to be uninstalled and run extra code; for instance, some samples (e.g., `5264`) open a web page. Interestingly, this technique is the one that comes closest conceptually to what we found in our results, and this shows again that allowing to monitor files within the installation path opens the door to state inference.

RQ6. Malware uses inotify in both Java and native code, but in most cases, we observed non-harmful operations, such as reacting to an app crash. We have encountered this phenomenon because malware is often produced by repackaging popular apps with malicious components. Therefore, we mainly detected the use of inotify by analytics libraries that can also be found in goodware. Only 4.2% of the samples try to monitor file system events generated by folders shared with other apps, but this behavior is again related to a famous advertising library. We found no samples installing an inotify watch on a file inside the installation folder of another app; in general, we found no cases suggesting the use of inotify for a state inference attack. However, we found again two interesting cases in native code where malware abuses inotify: an anti-debug technique and a "last resort" to execute code when they are uninstalled.

5.5 Analysis Methodology & Implementation

While comforting from an end user's perspective, the fact that malware does not leverage inotify for state inference attacks poses exciting questions. For example, have malware authors never come up with this attack idea, or do they instead not implement it because it does not work in practice?

While answering such questions on malware authors' behalf is rather tricky and far beyond the scope of this work, we can evaluate the feasibility of employing inotify for state inference at scale.

To this aim, this section introduces *inoTool*: a tool, written in about 2k lines of Python, that can prove whether an Android app is vulnerable to inotify-based state inference. *inoTool*

exposes three functionalities that capture and analyze an app's file system behavior, as discussed in the remainder of the sub-section.

5.5.1 Function #1 – *logEvents*

This function takes as input an APK file and computes the list of the file system events (from now on, we will refer to this list as *FSFootprint*) generated by the APK during its entire execution. A file system event is represented as the pair (*event_type*, *path*), where *event_type* is the inotify event corresponding to the file system action performed on the file at *path* (e.g., the event `IN_MODIFY` encodes a file content modification). Furthermore, *inoTool* stores the mappings between the app's package name and its *FSFootprint*.

At first, *inoTool* installs the app to analyze, and runs it for the first time without any monitoring. The idea behind the first "dry-run" of the app is to avoid capturing file system activities that are just the result of some post-installation setup operations. For example, an app that uses a SQLite database stored on the device's file system. The app likely creates this database only once, when it starts for the first time. If we considered the first run representative of the subsequent execution, we would wrongly expect it to create the SQLite file each time.

It is only while running the app a second time that *inoTool* collects the file system behavior of the app over 5 minutes. During this second run, *inoTool* also stimulates the app user interface, thus increasing its code coverage. For this purpose, it leverages ARES [181], a black-box tool that uses Deep Reinforcement Learning to test and explore Android apps.

The file system events monitor is developed as an eBPF program that hooks all the system calls – *syscalls* – related to file system events that can generate an event. Table 5.3 reports the complete list of monitored syscalls. eBPF (extended Berkeley Packet Filter) [122] is an in-kernel virtual machine that provides an interface to extend the capabilities of the Linux kernel with user-provided code. It has been available in Linux since version 4.4 and provides two advantages compared to adding new code directly in the kernel. First, the eBPF code can be updated without recompiling the entire kernel. Second, the eBPF VM code does not support potentially dangerous constructs (e.g., loops with non-fixed iteration counts) and undergoes a strict verification phase before execution, making it less prone to bugs. eBPF programs are event-driven: the eBPF code is executed only in response to specific events that include syscalls, making it a powerful tool for the syscall-based monitoring of Android apps.

The eBPF program is written in about $1,100$ lines of C code and leverages kernel *tracepoint* and *kretprobe* subsystem for monitoring syscalls. Tracepoints allow the eBPF

Table 5.3 Inotify event to system call

Inotify event	**System call**
IN_ACCESS	read execve execveat
IN_ATTRIB	chmod fchmod fchmodat utimensat setxattr lsetxattr fsetxattr chown fchown lchown fchownat utime utimes futimesat
IN_CLOSE_(WRITE|NOWRITE)	close
IN_CREATE	mkdir mkdirat link linkat symlink symlinkat bind mknod mknodat
IN_DELETE(_SELF)	rmdir unlink
IN_MODIFY	write truncate ftruncate
IN_OPEN	open openat
IN_MOVE_(SELF|FROM|TO)	rename renameat renameat2

program to intercept the beginning of each syscall and parse its input parameters. Kretprobes, instead, are used to monitor the return values and track the opened/closed files for each process, which is crucial because it allows *inoTool* to interpret syscalls that accept file

descriptors as input parameters. For instance, the first parameter of the `read` syscall is the file descriptor of the file from which to read. Our eBPF program keeps track of the mapping between file descriptors and file paths for each process in the system during the app under analysis runtime.

To start and stop the eBPF program each time a new app runs, we created a second program that registers/unregisters the kernel hooks and queries the eBPF program to retrieve the recorded events and save them into a JSON file. The second program is written in Golang (about 700 lines) and leverages the `libbpfgo` library [194] for interacting with the eBPF programs from userspace.

Finally, the function analyzes the syscalls recorded so far and converts them to a list of file system events (i.e., *FSFootprint*).

While, in most cases, mapping a syscall to an inotify event is straightforward (such mapping is reported in Table 5.3), three event types represent interesting exceptions: file closing, moving, and deleting. The first event type occurs when a process closes a file descriptor and distinguishes two cases. If the process had opened the file descriptor in writing mode, then an (`IN_CLOSE_WRITE`) event would arise; on the contrary, inotify returns an (`IN_CLOSE_NOWRITE`) event if the process did not open the file descriptor for writing. To correctly handle these two events, *inoTool* keeps track of the opening options for each file descriptor. The *move* event class distinguishes three different events. The first one – `IN_MOVE_SELF` – happens when the moved file is the program that spawned the process that moves the file. The other two – `IN_MOVE_TO` and `IN_MOVE_FROM` – arise when the file is moved *to* or *from* the monitored directory, respectively. Similarly, the *delete* event class depends on whether the watched file/directory was itself deleted (`IN_DELETE_SELF`) or the file/directory deleted from the watched directory (`IN_DELETE`).

5.5.2 Function #2 – *generateSignatures*

This function takes as input a set of apps K (cardinality $|K|$) and their respective *FSFootprint* generated by the *logEvents* function. It returns the $FSSignature_K$ for each app, that is, the set of file system events that our tool recorded ***uniquely*** for the startup of that APK; namely, it considers all the unique startup events of that APK's *FSFootprint* w.r.t. the other APKs in the set K. Notice that the choice of the apps in K determines how reliably an attacker can use the $FSSignature_K$. By tailoring K to the apps installed on the device, she targets this specific configuration to ensure that a file system event uniquely identifies an app's startup in that device.

Determining an app startup. Since our attack aims at identifying an app startup while constructing the $FSSignature_K$, we want to consider only those events generated during the startup phase. However, determining when an app startup ends presents several challenges, and today, it is still an open issue. Naive approaches like monitoring the screen until the user interface stops changing are not an option because apps showcase very different UI when they start (e.g., some have long-loading screens with animations, while others show advertisements). Relying on triggers in the app's code is also not feasible due to the heterogeneity in the UI frameworks Android apps use (e.g., Cordova, Xamarin, Flutter, WebView, etc.). For this work, we resorted to an experimental human-in-the-loop approach to determining when an app finishes loading. In particular, we randomly selected and ran 100 apps in our emulated environment while recording the emulator's screen. We then manually inspected the recordings and marked in each of them the time at which, according to our human understanding of the app's context, the startup process ended. The minimum startup time (rounded down) we measured was four seconds, which we then used to mark the end of the startup process for all the apps. We chose the shorter time conservatively to ensure we do not consider already started apps.

More precisely, the $FSSignature_K$ (S) of the n-th app is computed as:

$$S_n = F_n^+ \setminus \bigcup_{i \neq n} F_i \qquad \forall e \in F_n^+,\ t_e \leq 4\ seconds$$

where F_k represents the *FSFootprint* of the k-th app, and F_k^+ its subset that contains only the startup events (i.e., events that occurred in the first four seconds). It is worth noticing that to compute the $FSSignature_K$, from the F_k^+, we filter out all the events in the *FSFootprint* of the other apps in K, regardless of whether they occurred at the startup or during the apps' runtime. This procedure allowed us to avoid the case when a file operation performed by app A at its startup is also performed by app B at its runtime. Therefore, if an app has a non-empty $FSSignature_K$, monitoring one event of its signature is an excellent indicator to infer when it is starting up.

5.5.3 Function #3 – *signatureVerifier*

By construction, the $FSSignature_K$ of an app contains those unique events recorded during its startup only; i.e., the same events did not occur during other apps' execution. This means that an attacker capable of intercepting any event in an app's $FSSignature_K$ can mount an unambiguous state inference attack, using such events as an oracle.

This function, which takes as input an APK and its $FSSignature_K$, aims to assess whether it is indeed vulnerable to inotify-based state inference attacks or, in other words, whether an attacker can leverage an app's $FSSignature_K$ to mount a state inference attack. We accomplish this by mimicking an attacker's behavior using an *a posteriori* approach: we refine the $FSSignature_K$ using the events we can confirm that can be used in the attack. Thus, the result of this function is $FSSignature^*_K$, a subset of the $FSSignature_K$, which contains only the file system events that an attacker can monitor. In the end, if the $FSSignature^*_K$ is not empty, we declare such an app as 'vulnerable'.

We implemented our attack in an Android app we named *appSignVerifier* that attempts to intercept any event in an app's $FSSignature_K$. Under the hood, *appSignVerifier* uses inotify through the `FileObserver` class. Before registering an inotify watch, *appSignVerifier* checks whether the target file exists. If so, the watch is registered directly on the file. In contrast, if the file does not exist, *appSignVerifier* registers the inotify watch on its parent directory to intercept the file creation.

The entire verification stage adopts the following workflow. Similarly to *logEvents*, the *signatureVerifier* installs the app under test and starts it for the first time. It then provides the app's $FSSignature_K$ and installation path to the *appSignVerifier*, which sets an inotify watch on the events in the app's signature. *appSignVerifier* needs the installation path because events in the $FSSignature_K$ may be related to files in this directory; however, this path may change across two installations. The function starts the target app for a second time, allowing the *appSignVerifier* to collect the inotify events. After the app startup, it stops the *appSignVerifier*, which dumps the collected file system events on mass storage. Then, it prunes the list of recorded file system events of any entry not in the $FSSignature_K$. This allows filtering out those events that the *appSignVerifier* accidentally recorded while monitoring *any* file creation event in a directory. For example, if the $FSSignature_K$ of an app contains the path `/example/uniq.txt`, but this file does not exist before the app starts, *appSignVerifier* monitors the folder `/example/`, in which, however, other uninteresting file system events may occur. These events should, thus, be discarded.

At the end of the analysis, *inoTool* produces three sets of file system events for each app, for which the following relationship holds: $FSFootprint \subseteq FSSignature_K \subseteq FSSignature^*_K$. The cardinality of an app's $FSSignature^*_K$ equals the number of file system events in its $FSSignature_K$ that our *appSignVerifier* was capable of intercepting. Therefore, an app's $FSSignature^*_K$ is not the empty set if and only if at least one of its file system events arises during its startup only (that is, the same event does not take place during any other app's execution in a set K) *and* an untrusted app can effectively intercept such event.

5.6 Experimental Evaluation

This section presents the setup under which we tested a dataset of 4,863 Android apps using *inoTool*, as well as a preliminary and coarse-grained analysis of our findings.

5.6.1 Experimental Setup and Dataset

The analysis system is based on an Android emulator and runs Android version 12.0 (API level 31), equipped with the latest Google Play APIs, thus implementing all the latest security features. We equipped the emulator with a dual-core 2.10 GHz x86-64 processor, 2 GB of RAM, 32 GB of internal storage, and a 16 GB emulated SD Card. To execute the eBPF program, we had to recompile the Android Linux kernel from the source code to enable the `CONFIG_KPROBES*`, `CONFIG_BPF_SYSCALL`, and the `CONFIG_IKHEADERS` configuration options.

We built the dataset of apps to test by collecting Android applications that are valuable and widespread among Android users and that implement a relatively heterogeneous set of functionalities. The first two characteristics make these apps appealing targets for attackers, which may profit from a vast user base. On the other hand, the variety of functionalities ensures that the app we analyze shows different behaviors. This prevents our results from being skewed toward one particular type of app and more representative of the average user's device. As reported by Google [93], Android users install, on average, at most 35 apps spread over different categories.

We collected the 100 most downloaded free apps for the 50 categories in the Google Play Store [96]. Some of these categories are particularly valuable, especially for phishing attacks, such as finance and social media apps. In total, we could install 4,863 apps because some of them were incompatible with our emulator.

5.6.2 Preliminary Analysis

Table 5.4 Cardinality statistics with $|K| = 4{,}863$

Set	Min	Max	Avg	Stdev	Median
FSFootprint	17	3087	463.2	192.1	457
$FSSignature_K$	3	784	65.1	74.2	28
$FSSignature^*_K$	2	61	6.8	2.6	7

Table 5.4 shows aggregate data about the *FSFootprint*, $FSSignature_K$, and $FSSignature^*_K$ that *inoTool* generated for each app.

On average, the cardinality of an app's *FSFootprint* is seven times that of its $FSSignature_K$. As one may expect, most of the file system activity during an app's startup does not depend on the app itself but rather on the implementation of the app startup process. For example, to bootstrap any apps, the Android system must load and read the `/system/framework/x86_-64/boot-framework.art` and `/apex/com.android.art/javalib/x86_64/boot.art` files, which contain the ahead-of-time compilation result of some framework classes. Nonetheless, roughly one in four file system events are unique to the app, leaving room for inferring which app is starting based solely on the file system activity.

While significantly (roughly ten times) smaller than the $FSSignature_K$, the $FSSignature^*_K$ of an app contains, on average, 6.8 file system events. As a reminder, these are file system events that are not only specific to the app but can also be monitored by a third-party application using the inotify system. To our surprise, the lowest cardinality among the computed $FSSignature^*_K$ is two, meaning that ***all*** the apps in our dataset have a non-empty signature and are susceptible to inotify-based state inference attacks.

Upon further analysis, we found some file system events common to the startup process of all the apps we tested. Since they stem from the app startup process implemented by the Android system, we define these events as *system-dependent*.

An example of a system-dependent file system event is the opening of the `base.apk`, located in the app's installation directory. It is accessed by the Android OS to retrieve its resources (e.g., the Manifest), resulting in an `IN_OPEN` and an `IN_ACCESS` file system events. Our analysis demonstrated that *any* program in the system that knows an app's installation path could intercept these two events on such a file. This is possible because of this file's permissive access control rules. From a DAC point of view, any app's `base.apk` is world-readable, while SELinux MAC rules explicitly allow it.

Other system-dependent events concern the `base.odex` and `base.vdex` files, also located in the app's installation path. These files are the byproduct of the ahead-of-time compilation process implemented in the Android RunTime that translates the app bytecode into machine code at installation time. As such, the Android framework has to access and load these files in the address space of the newly spawned app; in fact, the former contains the machine-compiled code, while the latter stores data designed to speed up the execution.

App-dependent events represent the other side of the coin compared to system-dependent ones. These file system events are tightly related to an app's behavior rather than the implementation of the bootstrap process in the system framework. App-dependent events

may concern files located anywhere on the file system. Table 5.5 presents the distribution of *app-dependent* events of our dataset throughout an Android file system. Specifically, each row in the table represents a path in the Android file system, defined in top- and second-level (first and second column, respectively) directories. The third column provides the share of apps in the dataset whose $FSSignature^*_K$ contains at least one event in the corresponding directory.

Table 5.5 Path distribution of files in $FSSignature^*_K$

Root	Depth=1	Percentage %
/data		17.20
	/app	17.20
/storage		1.5
	<sdcard_ID>	0.68
	/emulated	0.84
/sys		0.06
	/devices	0.06
/system		0.03
	/priv-app	0.03
	/lib64	0.02
/etc		0.02
	/passwd	0.02
/product		0.02
	/overlay	0.02

Although not as common as their system-specific counterparts, the most common location of app-dependent events is still the app's installation directory (/data/app row in Table 5.5), in which 17.2% of apps generate at least one event. Examples of such events are those related to native libraries shipped with the app and placed in the lib directory in the installation path.

1.5% of apps generate at least one event in the /storage/ top-level directory – also known as *external storage* – which contains app-specific sub-directories where apps can place their persistent files. Notably, external storage comprises two parts. The first is a mount point at which the Android system mounts external mass memory devices (e.g., SD cards), which account for roughly 45% of the events *inoTool* detected in the external storage. On the other hand, the second is not backed by any external memory device and is created by the system to provide external storage functionalities without external memory support. This second part accounts for most of the events *inoTool* reported in external memory.

File system events in other subdirectories are generally less common and affect less than 1% of the total apps in our dataset. Nonetheless, an attacker can leverage them to mount for state inferencing purposes. Interesting examples of events in these paths concern

the `/system` top-level directory. A peculiar case is that of an app that interacts with the `/system/priv-app/InputDevices/InputDevices.apk` file to check if a particular input device is available.

5.7 Attacker Models & Attack Scenarios

Analyzing the results obtained by testing the 4,863 apps through *inoTool*, we identified three distinct sets of capabilities that an attacker needs to perform an inotify-based state inference attack. In this context, a *capability* may be an Android permission or information about the installed apps. In general, obtaining a specific capability may depend on I) the version of the Android system *and* II) the API level that the malicious app targets. The impact of the former on the attacker's capability is relatively intuitive: different versions of Android implement slightly different functionalities and security models. To fully appreciate the latter's role, one needs to consider how Android deals with the app's backward compatibility. At compilation time, an app developer can choose which version of the Android framework the app is intended to run. If the targeted version is "too old," the OS will refuse to install the app. On the other hand, if the targeted version is still supported, the system emulates the targeted version's behavior and security model. For instance, more than 68% of the apps in the dataset target API 30, even if, at the time of writing, the latest Android version is 13 (API 33). As such, most of the apps we analyzed are not subject to the latest and more restrictive security model.

In the remainder, this section will assume that the attacker has paved the way to install her malicious app on the victim's device – a basic premise for any state inference-based phishing attack. While this may happen in various ways and represent an exciting research field in its own right, we will only develop this topic further as we consider it orthogonal to the research questions this work addresses. However, we assume the malicious app to be a regular Android app, thus subject to the Android security model for untrusted apps with restrained access to the device's resources.

5.7.1 Attack #1

As we saw in Section 5.6, while starting any app, the Android system accesses the app's specific files (e.g., the `base.apk`) in its installation directory. In other words, an attacker able to monitor those files can target *any* app on the system. Luckily, an app's installation path is not known a priori (part of its name is randomly generated at installation time), and,

at least in theory, it can only be retrieved through the Package Manager service (as discussed in Section 2.1.1). Before Android version 11, interacting with this service did not require any permission, allowing all apps on the device to access each other's installation path. To limit Software Discovery [45] (i.e., listing installed apps and their metadata), Android 11 introduced the concept of *package visibility* [97]. In practical terms, under the new security model, an app must either list the metadata of all third-party apps it needs to access (through the <queries> tag in its Manifest) or request the QUERY_ALL_PACKAGES permission. From a malicious actor's perspective, both mechanisms could be better. For what concerns the first option, an app whose Manifest targets many apps is rather suspicious and incurs higher risks of being banned by the marketplace. Similarly, requesting the QUERY_ALL_PACKAGES permission is not a sustainable plan for an attacker. Google states that any app requesting this permission must undergo manual scrutiny before being accepted on the Google Play Store [97].

However, there are two loopholes that a malicious app can leverage. The first takes advantage of Android's backward compatibility system. The package visibility mechanism does not apply to apps targeting API level 30 or lower, which can retrieve the installation path of *any* app in the system without any permission. The second technique consists in (ab)using one feature of the <queries> tag that gives access to any app that implements a particular *intent filter* [87], as shown in Listing 5.1.

Listing 5.1 queries tag to interact with all applications

```
1 <queries>
2   <intent>
3     <action android:name="android.intent.action.MAIN"/>
4     <category android:name="android.intent.category.LAUNCHER"/>
5   </intent>
6 </queries>
```

This allows querying *all* the apps with a "launchable" activity – i.e., those presenting one entry in the Android launcher UI – which is the case for practically every commonly used app. While conforming to the documentation, this stratagem contradicts the app separation principle at the core of the Android security model. We also reported this bug to Google, which acknowledged its security implications and said the issue had already been reported but has yet to be made public.

5.7.2 Attack #2

The second attacker model we envisage assumes that the loopholes we introduced in Attack #1 are no longer available, making monitoring the apps' installation path impossible for an attacker.

In this new scenario, the attacker chooses a dataset of large and heterogeneous apps and similarly analyzes them to that presented in Section 5.6, embedding each app's $FSSignature^*_K$ in her malicious app. In analyzing what an attacker can achieve by implementing this generic strategy, we will refer to the data we collected using *inoTool*. The files in the $FSSignature^*_K$ generated in our experiments can be divided into two categories: those that can be accessed via the `READ_EXTERNAL_STORAGE` permission and those that are world-readable, i.e., they do not require any permission to be monitored.

READ_EXTERNAL_STORAGE permission. Earlier Android versions used this permission to restrict access to external storage. Android 10 (API 29) introduced a new file system paradigm called *scoped storage* [88] that divides the external storage into private and shared portions. As the names suggest, while the former is app-specific, the latter contains data shared across all the apps in the system and is further divided into media and non-media content. The new paradigm redesigned the `READ_EXTERNAL_STORAGE`, allowing an app to read only other apps' shared media content. Consequently, obtaining the permission in question drastically changes the benefit an attacker gains. While on older versions of Android (API < 29), this permission allows identifying apps that access any file in the external storage, on newer versions, it only allows detecting those that access shared media files.

Based on our dataset, this discrepancy translates into roughly ten times fewer apps being vulnerable while running in Android 10 or newer (0.7% of the dataset) w.r.t. older versions (18.1%). These figures do not include world-readable files (discussed below), so they can be considered the permission's contribution.

World-readable files. We discovered something peculiar by analyzing the world-readable files *appSignVerifier* managed to exploit. While on paper, files in an app's "private" storage are accessible exclusively by the app itself, in practice, it is still possible to monitor some of them through *inotify*. In particular, our analysis shed light on the fact that apps' private folders in the *portable* configuration of an SD card could be monitored using the inotify API (contrary to what an attacker can do in the internal memory).

To make things worse, unlike the installation directory, an app's private storage path does not contain any randomly generated part, making it trivial to infer. In other words, in a bizarre turn of events, the scoped storage's implementation breached the tight access control

that regulated the external storage in previous versions of Android. In our dataset, 0.9% of apps are vulnerable to state inference attacks that do not require any particular capability from the attacker. Among such apps, we find two famous instant messaging (WhatsApp and JioChat) and a money transfer (Venmo) app. They have large user bases, having been downloaded at least 10M times each; therefore, unauthorized access to user accounts on these platforms may have severe privacy and financial repercussions.

5.7.3 Attack #3

The main advantage of the Attack #2 attack strategy is that the pre-computed $FSSignature^*_K$ can be embedded in a single malicious app and shipped *as is* to the victim's device. Moreover, it represents the most generic attack possible. This strength, however, comes at the expense of overapproximating what the vulnerable apps are on an actual device. When computing the file system signatures on a dataset K as large as the one we used in our experiment, the attacker implicitly assumes that *all* the apps in the dataset could be installed at once on the targeted device. This is different for the average Android device that, according to Google [93], tends to have around 35 apps installed at a time.

Intuitively, narrowing down the cardinality of K increases the number of apps with a non-empty $FSSignature^*_K$. The sweet spot for an attacker is to choose K as the set of apps installed on the target device. Doing this maximizes the number of apps vulnerable to state inference attacks *on that specific device*.

Thus, the third attack strategy consists of scouting the apps installed on the infected device, computing the apps' $FSSignature^*_K$ accordingly with the aid of a remote endpoint, and employing them to perform state inference attacks.

The sole capability the attackers must acquire to implement this strategy successfully is obtaining the list of installed apps on the device. Similarly to what is described in Section 5.7.1, nowadays, the most trivial way to do that would be through either the `QUERY_ALL_PACKAGES` permission – which is too suspicious to be considered a viable option – by defining the `queries` tag or targeting API level 29 or below. One last option at the attacker's disposal is leveraging an information disclosure vulnerability, like the one we discovered in our research.

Information Disclosure Vulnerability. The introduction of scoped storage loosened the access control on external storage, allowing otherwise unprivileged apps to mount state inference attacks. In particular, we noticed that the paths where each app saves its private files are predictable, following the pattern `<external storage path>/Android/data/-`

<package name>. For example, on our test device, the private files of the famous Discord app (whose package name is `com.discord`) were located in the `/storage/emulated/-0/Android/data/com.discord/` directory. Unlike the app's installation path, the private storage path contains no aleatory component, making it trivial for an attacker to infer. Simply invoking the `getExternalStorageDirectory` API (which does not require any permission), an app can retrieve the <external storage path>. In contrast, the target app's package name can be determined by statically analyzing its APK available on the marketplaces.

What makes private storage vulnerable, however, is the fact that the app's files are at predictable locations *and* the permissive access control rules that the system applies to them are not enough. The `data` directory in the external storage is marked as not readable, not writable, but executable for any untrusted app. Under the POSIX convention, this combination of flags denies anybody but the directory's owner and group to list its content and create files. However, the executable attribute allows anybody to *traverse* the directory. Therefore, an untrusted app – even *without* the `READ_EXTERNAL_STORAGE` permission – can check if a specific file exists in the `data` directory using of the `newfsstat` syscall or, more conveniently, through the `java.nio.file.Files.exists` Java method. More specifically, when querying for an existing file, the syscall (and the wrapper) returns successfully. To an extent, this primitive can circumvent the lack of listing privilege on the directory. Thus, an attacker can maintain a list of package names of apps they want to target and systematically probe each of them.

Suppose such a folder is in the `data` directory. In that case, the attacker is sure that the corresponding app is installed on the device because all its private folders are removed whenever an app is uninstalled. In addition, the system creates the app's private folder on the external storage whenever it invokes one of the several methods to retrieve its private location for the first time, such as `android.content.Context.getFilesDir` and `android.os.Environment.getExternalStorageDirectory` methods. Table 5.6 reports the complete list of such methods and the prevalence in our dataset of benign apps.

By statically analyzing the apps in our dataset, we discovered that almost the entire dataset ($> 99\%$) imports at least one of these methods. In particular, the `android.-content.Context.getFilesDir` method is the most prevalent and can be found in 98.7% ($4,632/4,863$) of the apps in our dataset. However, the static analysis does not guarantee that every one of these apps invokes any of these methods. This result should be considered an upper bound of the share of apps that use external storage and are thus vulnerable. To

Table 5.6 Android APIs usage to interact with the external memory.

Path	**%**
`android.content.Context.getFilesDir`	98.73
`android.os.Environment.getExternalStorageDirectory`	97.15
`android.content.Context.getExternalFilesDir`	91.35
`android.content.Context.getExternalFilesDirs`	90.34
`android.content.Context.getExternalCacheDirs`	90.06
`android.content.Context.getExternalCacheDir`	88.10
`android.content.Context.getExternalMediaDirs`	87.07
`android.content.Context.getObbDirs`	74.73
`android.content.Context.getObbDir`	61.96

estimate a lower bound, we measured the number of apps for which *inoTool* registered a file system event in the corresponding private folder, and we obtained 49% (2,383/4,863).

In conclusion, we can estimate that 49-99% of the apps in our dataset can be detected on a device without permission. This information disclosure vulnerability partially thwarts Google's efforts to restrict third-party apps from Software Discovery.

Performance of the refined attack. To estimate the likelihood of success of an attack tailored to the apps installed on a device, we relied on random sampling apps from our dataset and computing the number of vulnerable apps in each case. For each simulation, each app's chance to be selected depends on the downloads from the Play Store, so the ratio between the two apps' download counts equals the ratio of their chances to be chosen. For example, the eToro and FlyerMaker apps have been downloaded by 10M and 1M users, respectively; thus, the former is ten times more likely to be selected than the latter.

We varied the cardinality of the sampled apps from 15 to 100, increasing it by five each time. We created 5,000 unique app sets for each cardinality, i.e., 5,000 sets of cardinality i. For each set, we executed the *generateSignatures* and *signatureVerifier* functions of *inoTool* to measure the number of vulnerable apps to our attack with this particular device configuration.

Figure 5.1 shows the results of this experiment, divided according to the permissions and Android version required to monitor files with inotify. The lines are averages of the percentages of vulnerable apps, while the faded areas are the standard deviation.

As expected, the number of vulnerable apps decreases with the number of apps installed. Alarmingly, on devices with an average number of installed apps (35, according to Google [93]) running modern Android versions, an attacker can successfully target half of the apps, even without declaring any permission. While performing slightly better when targeting API level < 29, under modern security models, the `READ_EXTERNAL_STORAGE`

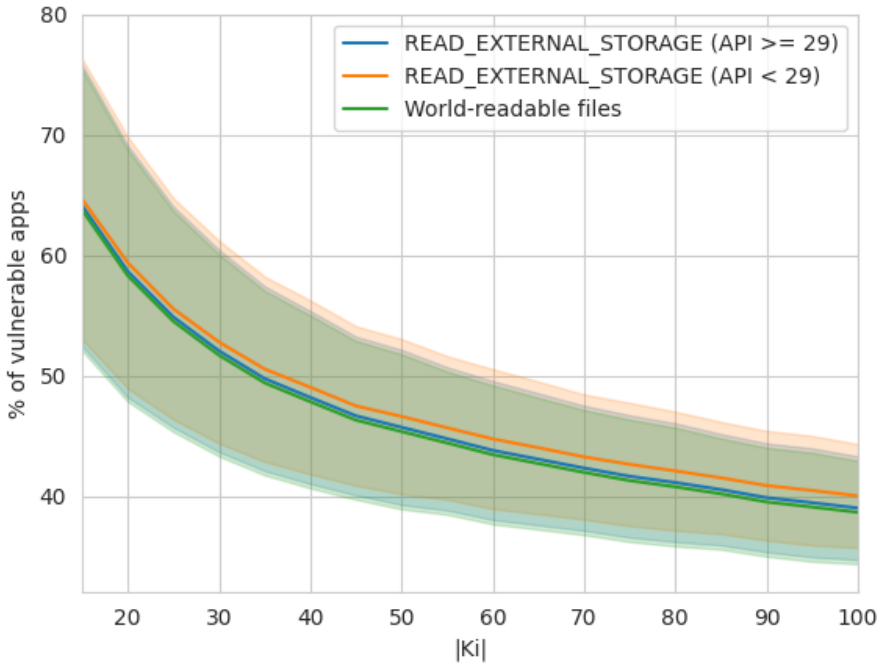


Figure 5.1 Capability for different K_i values

does not significantly affect the likelihood of carrying out the attack w.r.t. to targeting world-readable files only.

5.7.4 Final Considerations

The following is a summary answering **RQ7** and **RQ8**. We identified three types of attacks, which differ according to the attacker's capabilities. If an attacker can monitor a system-dependent file system event (Attack #1), the opening of the `base.apk` is the best candidate. To this aim, she needs to know the installation path, and the only ways to get it are: by requiring the `QUERY_ALL_PACKAGES` permission, abusing the `queries` tag, or targeting SDK ≤ 29. This attacker is the most powerful because it works with 100% of apps regardless of the Android version and the device configuration.

In case an attacker cannot retrieve the installation path, she can pre-compute a $FSSignature^*_K$ with a large K analyzing as many apps as possible to find peculiar behaviors that enable an attacker to perform an inotify-base state inference attack (Attack #2). Our evaluation with $|K| = 4,863$ found that an attacker can target 0.9% of the app by monitoring only world-readable files. Obtaining the `READ_EXTERNAL_STORAGE` permission, the number of

vulnerable apps increases to 1.6% and 18.8% for Android 10 or newer and older versions, respectively. However, these values also consider the contribution of the world-readable file since an attacker can always monitor them. Thus, the actual contribution of this permission is 0.7% on Android 10 or newer and 18.1% on older versions. Despite these low values, this is the most general scenario: an attacker can always attempt to use this approach to search for peculiar events in the *FSFootprint* of the target app.

Attack #3 is a refinement of the previous attack strategy that an attacker can adopt if she can get the list of installed apps, e.g., by using the information disclosure vulnerability we reported. Our experiments suggest that in a typical device with 35 apps installed on average [93], the attacker can target almost half of the apps without any permission.

Finally, the Android version largely influences the number of vulnerable apps and the attacker's capabilities. First, the metadata of the installed apps (e.g., the installation paths) can be obtained without permission on a device running Android < 11; on Android ≥ 11, an attacker has to update the Manifest file appropriately. Second, the number of apps vulnerable requiring the `READ_EXTERNAL_STORAGE` permission strongly depends on the version of the operating system.

We summarized the workflow of the different attacks in the flowchart in Figure 5.2, and for each case, we report the percentages of the vulnerable apps in our dataset. The percentages reported for Attacker #3 consider the average case of the typical device with 35 apps installed, as described above.

5.7.5 Use Case with Contextual Notification

To demonstrate the impact of an inotify-based state inference attack, we developed a toy app implementing a full zero-permission phishing-like attack targeting the eToro (package name: `com.etoro.openbook`) app. According to the Play Store, this app counts tens of millions of installations and provides access to a trading platform, making it the perfect target for a malicious actor.

The $FSSignature^{*}_{K}$ we computed for this app contains the file system events for opening (`IN_OPEN`) and reading (`IN_ACCESS`) the `/system/framework/android.test.runner.jart` file. The corresponding file is accessible by any app on the system, including unprivileged apps, without permission.

Figure 5.3 shows the attack flow in three steps. Our malicious app registers an inotify event at the beginning of its execution, using the `FileObserver` API. When the user launches the eToro app (Step 1 in Figure 5.3), the inotify callback is triggered. At this point, our

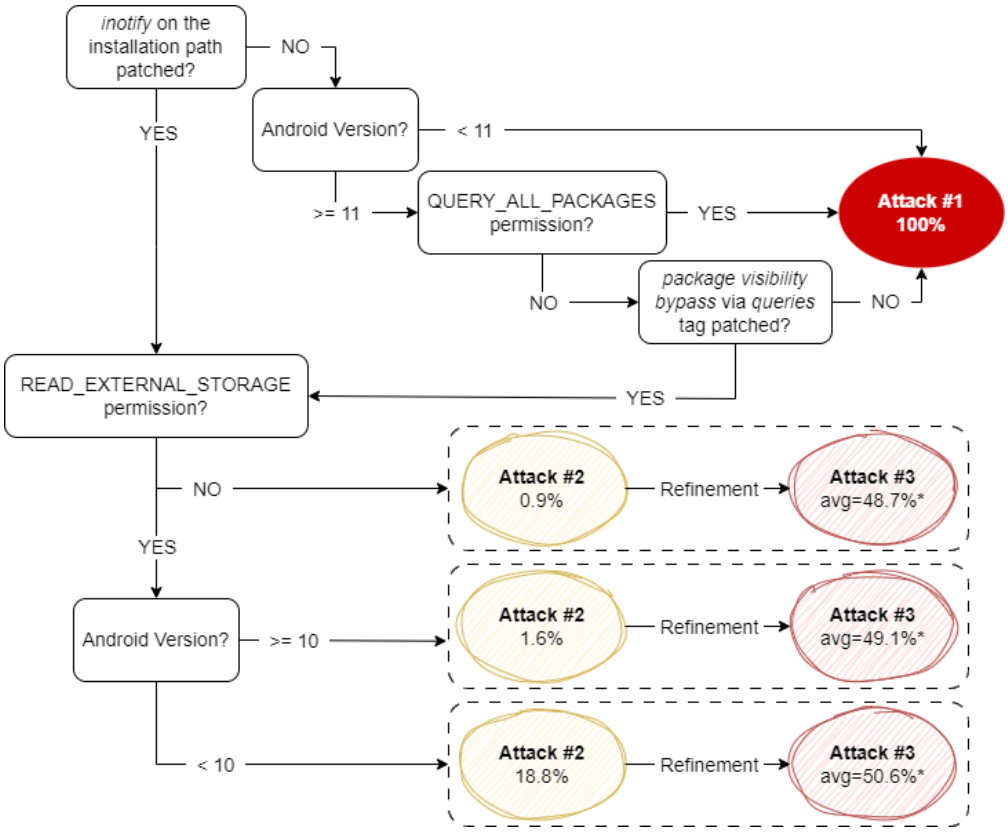


Figure 5.2 Flowchart summarizing the attack models of inotify-base state inference attack.

malicious app does not immediately create an activity that mimics the eToro's one as it does not have the needed permission. In the modern Android system, starting an activity without the user's interaction is strictly regulated since this type of action has long been used to implement UI hijacking. The only viable way to implement such behavior is by obtaining the `SYSTEM_ALERT_WINDOW` permission, which has been ranked as *privileged* (refer to Section 2.1.1) after previous works demonstrated its malicious potential [71]. This fact did not stop malware authors from using it; in fact, 56.9% of the samples (5,692/10,000) among the malware we used in this study (Section 5.4) require it, while in the case of benign apps (Section 5.6) 8.6% (431/4,863). However, implementing our phishing attack using this permission differs from our purpose of showcasing how a complete permission-less phishing attack could work.

Xu et al. already proposed using the notification service on a mobile phone to launch phishing attacks [241], and we decided to refine this idea. Instead of requiring the abovementioned permission, our malicious app opts for displaying a *contextual notification* (Step 2), trying to lure the user into initiating a fake update process for their eToro app. A contextual

Figure 5.3 Flow of the attack with a contextual notification

notification, while unexpected, is not necessarily suspicious from the user's perspective. In this case, the app may have found an available update while loading and asked permission to install it.

Since it mimics the target's look&feel, it is tough for the user to distinguish our malicious update prompt from a regular eToro notification. Step 2 also shows how legitimate and phony notifications render when the malicious app targets different Android operating system versions. In the first case targeting an older version of Android, the forged notification is practically impossible to distinguish from a legitimate one because the attacker has complete control of the notification's UI. While targeting Android 12, instead, the user's only chance to understand that the notification is not genuine involves expanding the notification. Starting with Android 12 (API level 31), apps can no longer create fully custom notifications [91], and the system applies a standard template instead. As the Figure shows, under these circumstances, the user can read the name of the apps that generated the notification – which is, nevertheless, attacker-controlled, as we emphasize in our example.

Once the user clicks on the contextual notification (Step 3 in Figure 5.3), our malicious app can effectively take control of the UI system and finally create the activity that asks the user for their credentials.

RQ9. During our analysis, we discovered several loopholes that grant an attacker different capabilities to perform a state inference attack and mount a practical phishing attack. Moreover, we highlighted that a malicious actor can still perform this attack without particular capabilities by monitoring world-readable files and presenting the user with a contextual

notification. An attacker can mount the attack assuming that she controls an untrusted app with no permissions, as we showed with our malicious app.

5.8 Mitigations

Even if this work demonstrated that our attack also works in case the attacker cannot monitor files in the installation path, this capability is the cornerstone of our work that Google has acknowledged. Therefore, it is evident that the primary fix is to avoid untrusted apps from watching files in the installation path, even on their own, because we have also noticed malware abusing it to execute code when uninstalled. To propose a solution to this vulnerability, it is imperative to understand the root cause that allows an attacker to exploit it. Given that Android uses SELinux to enforce MAC, which uses a "default-deny" approach, as explained in Section 5.2, there must be rules that explicitly allow monitoring of the installation folder.

Watching the installation path. As a first step, we inspected the file where SELinux kernel hooks are defined to determine whether there was any filtering related to inotify syscalls [84]. We identified that SELinux sets up the `selinux_path_notify` hook for handling these syscalls. In particular, the hook verifies if the context of the subject contains the `watch` and `watch_reads` SELinux permissions. The difference between the two lies in the fact that `watch_reads` is only related to read-like events.

Then, we proceeded to check if any SELinux rules would allow a third-party app (running in the context of `untrusted_app`) to use inotify (i.e., allowing `watch` or `watch_reads`) on files or subfolders of the `/data/app/` (which are labeled as `apk_data_file`). It is worth noting that, on Android, the `untrusted_app` context inherits all the `appdomain` rules. After retrieving the compiled SELinux policies from our emulator (namely, the `/sys/fs/selinux/policy` file), we searched for these rules, and our hunch turned out to be correct. We found two rules that allow the `appdomain` (thus, `untrusted_app`) to use both `watch` and `watch_reads` on files and directories labeled as `apk_data_file` (i.e., files and directories in `/data/app`). A deeper analysis allowed us to identify how both `watch` and `watch_reads` permissions are defined in the `r_file_perms` and `r_dir_perms` macros [90].

As mitigation for this vulnerability, acting at the SELinux level is the best choice as it ensures a strong level of security and controls its granularity simultaneously. For example, removing the POSIX world-readable permission on the `base.apk` would be wrong. A typical use case is antivirus software, which should be able to read this file for its analysis. Therefore, we propose to create a new macro (e.g., `r_file_perms_nowatch`) with the same

permissions of `r_file_perms` but without `watch` and `watch_reads`. Then, this new macro can be used to define the permissions of `appdomain`, and thus consequently `untrusted_app`, for operating on `apk_data_file`. This way, we can prevent an untrusted app from listening for inotify events on files or directories labeled `apk_data_file` without limiting access to this file. However, we can always allow third-party apps to use inotify, for instance, on files or directories belonging to their sandbox, which are legitimate uses and scenarios. This mitigation effectively eliminates the main vulnerability we found while not compromising the stability and usability of the system and introducing a highly fine-grained level of control on which processes can use inotify and where.

We implemented this mitigation by modifying the AOSP source, and our emulator works correctly but without allowing monitoring files within the installation path with inotify. However, only Google has the resources to test if this change negatively impacts the whole ecosystem.

Third-party vendors – RQ10. Since Possemato et al. [171] have shown a significant difference in the security posture of AOSP concerning the Original Equipment Manufacturers, we investigated if their flagship smartphones allow using inotify on the installation path.

We collected the major vendors' latest ROMs – Samsung, Xiaomi, Motorola, Lenovo, and Realme – from `stockrom.net` [208] or `firmwarefile.com` [66] (see Table 5.7), resulting in a dataset of 11 images. We collected only the ROMs with the latest security upgrades (e.g., scoped storage) to gather reliable results, preferring Android 12 when available. Then,

Table 5.7 ROM versions & models

Brand	Model	Vulnerable
Samsung	Galaxy S20 Ulta	✓
	Galaxy S10e	✓
	Galaxy A23	✓
Xiaomi	Xiaomi 12S Pro	✓
	Redmi Note 8	✓
	Redmi 10	✓
Motorola	Edge 20 Pro	✓
	Moto G52	✓
	Moto G30	✓
Realme	C25 RMX3191	✓
Lenovo	Legion Y90	✓

we extracted the content of each ROM, looking for the SElinux policy files. However, the `/sys/fs/selinux/policy` file is available only at runtime; thus, we searched for the

source file (that changes depending on the customization; examples are `sepolicy`, `file_contexts`, and `property_contexts`). The analysis allowed us to identify that all the tested ROMs inherit the AOSP SELinux policy, which assigns to an untrusted app the `watch` and `watch_read` permissions. Thus, *all* the tested ROMs are also vulnerable.

Information disclosure. Then, the second most common paths found in the $FSSignature^{*}_{K}$ (Section 5.6) and the information disclosure vulnerability (Section 5.7.3) are related to the external storage (i.e., `/storage/`). These vulnerabilities are easily solvable with POSIX permissions. All files and folders in the private external storage should be read-write-execut-able by the owner. At the same time, to solve the information disclosure, the `Android/data` folder in the emulated storage must not be world-executable. Moreover, to err on the side of caution, it would also be appropriate to introduce a random string in the presence of folders with the package name of the app (e.g., `/storage/emulated/0/Android/data/com.discord/`), as is already the case in the installation path.

RQ11. There are mitigations, and Android system developers should act in combination with SELinux and POSIX permissions. SELinux policies should be extended to prevent an untrusted app from installing inotify watches in any installation path, and only the owner should be able to operate on her private files in the external storage. Moreover, similarly to the installation path, folders named with the package name of an installed app can lead to information disclosure or path traversal; therefore, they should be accompanied by a random string. However, we argue that comprehensive remediation does not exist and Attack #2 can always be attempted (e.g., through a world-readable file). As in the Man-in-the-Disk, the $FSSignature_{K}$ of an app depends on its specific behavior in combination with one of the other installed apps, thus resulting in a complex environment that is difficult to defend.

5.9 Discussion

Limitations. This work is not exempt from limitations.

First, the timespan considered as the startup time of the app crucially determines the events in the $FSSignature_{K}$ and, thus, the attack's impact. If the timespan is too short, potential events are lost, limiting the attack; if it is too long, the app may have already started when the attack takes place, and the user may notice odd behaviors.

Second, we collected the file system events stimulating each app with ARES. Thus, we inherit the limitations of the dynamic analysis [5, 169]: such traces may not be complete because the analysis did not lead to specific code.

Third, by exploiting evasive controls, some apps' components (e.g., ads libraries) may show different behaviors depending on the device on which they are executed (e.g., actual device vs. emulator). However, advertisement libraries are part of a shared codebase that would likely not generate unique events; moreover, these evasive checks are typical of malware.

Fourth, in our analysis, we dwelt on monitoring a single file, and given its effectiveness, we explored no further; Ahn et al. [6], instead, focused on a sequence of events. Although there is a margin for improvement, our work is mainly concerned with showing the existence of this new attack and how to stop it. Future works must consider the actions Google will take after our disclosures.

Google Play Store. This chapter has always referenced the Google Play Store's policies and how they may limit the proposed attacks. In particular, we emphasized that if an app that requires the `QUERY_ALL_PACKAGES` permission (thus, looking for the installation path) is uploaded to the Play Store, the app's use of this permission is subject to approval based on specific security policies. Even assuming Google fixes the `queries` tag workaround, the level of such permission is "normal" (i.e., automatically granted at installation time); therefore, there are two crucial factors to consider.

First, all the restrictions Google imposes on its store have led malware authors to refrain from uploading their malicious apps directly. They upload droppers to side-load the malicious app with extended permissions [146, 207, 221]. Therefore, in such a scenario, our attack has no limitations.

Second, the Play Store is one of the many app distribution channels. For instance, in China, the Google Play market share is less than 4%, while MyApp (Tencent) [219] currently holds 25% of the market. Therefore, such a plethora of alternative markets (e.g., Samsung Galaxy App, Amazon App Store, AppBrain, etc.) implies that our attack tremendously impacts the global Android ecosystem.

Responsible Disclosure. We have reported three vulnerabilities to Google through the official issue tracker:
I) The possibility of using inotify in the installation folder (reported in May 2022, acknowledged the same month);
II) The information disclosure, described in Section 5.7.3 (reported in July and acknowledged in September 2022);
III) The package visibility bypass via the `queries` tag, described in Section 5.7.1 (reported in July 2022).

The first two were acknowledged as bugs of Moderate severity [89], while other researchers previously reported the third, but it was not public when we developed this work.

Chapter 6

A Comprehensive Analysis of Android Evasive Behaviors

6.1 Introduction

Given that Android malware has significantly evolved in terms of its capabilities, sophistication (e.g., they are increasingly abusing and exploiting the native layer as discussed in Chapter 4), and adoption of evasive techniques [187, 128] over the past years, this chapter studies the anti-behavioral mechanisms used by modern Android apps. In particular, this work focuses on techniques for detecting different forms of dynamic analysis (i.e., anti-behavioral techniques discussed in Section 2.2.2). While these techniques are prevalent among malware to avoid exposing the malicious behavior inside an analysis environment, they are also adopted by benign apps to protect their code from reverse engineering and specific client-side attacks or to ensure that the users' sensitive data (e.g., bank access tokens) are not stored in a rooted device [149, 196, 29]. Either way, evasive controls are part of the surreptitious code's techniques, and their goal is to protect the apps – whether benign or malicious – by retrieving precise and accurate information on the hardware and software components of the system they are running on.

Dealing with this topic is as vast as it is complex. First, this chapter aims to study known evasion techniques without attempting to detect new ones or look for those still unknown (i.e., the evasive techniques introduced and categorized in Section 2.2.2). Moreover, we proposed a novel categorization into two main groups based on the techniques' implementations, namely *direct* and *indirect* evasive techniques. *Direct* evasive techniques (DETs) retrieve specific data that can be directly used to detect whether the app is executed inside an analysis environment. Conversely, *indirect* evasive techniques (IETs) return data that must be further processed

to be used for evasion. This distinction is crucial for detecting evasion attempts: a sample employing DETs is unequivocally looking for information on the runtime environment, which per se is enough to infer an evasion attempt, whereas merely engaging IETs may serve the same objective, but not necessarily.

Therefore, we studied all possible implementations and developed a proof-of-concept Android app implementing all the collected evasive techniques we gathered (the app is available at [15]). We were inspired by Al-Khaser [135], an executable for the Windows OS developed to test the stealthiness of sandboxes, which has also been used in several scientific papers to study the Windows evasive malware [55, 143, 75] phenomenon.

Then, a sandbox to execute benign and malicious samples is needed to measure the techniques used in the wild. In the context of malware analysis, the term 'sandbox' generally refers to a dynamic analysis tool that runs in a safe and controlled environment for analyzing and observing suspicious and surreptitious code behavior without risking damage to the host or the network [118]. Sandboxes may offer several advantages, such as quickly restoring the environment after analysis, ensuring that any malicious activity stays confined to the sandbox, scalability, and replication for a wide range of configurations, making testing the behavior of several suspect samples across different scenarios easier. In [32], the authors highlighted the requirements that a malware analysis sandbox for Android should follow. They can be summarized in two criteria: *anti-evasion* (or "resilience to the detection") and *maintainability*. The former refers to the ability of a sandbox to be transparent to the apps running inside it, creating the minimum possible set of artifacts that allow an app to detect the sandbox itself. Ideally, a sandbox exposes realistic and consistent information; otherwise, malware may leverage non-coherent knowledge to build a novel evasive technique. Maintainability quantifies developers' effort to address new evasive controls and upgrade the sandbox with a new version of the Android (kernel and operating system). Achieving maintainability does not necessarily require avoiding kernel and OS modifications as long as they are easily portable to the newer versions. Another critical aspect of large-scale malware analysis is *scalability*, which measures the system's ability to analyze many samples simultaneously in an automated fashion and under several device configurations.

In Android, sandboxes are often implemented as emulators [243] or as containers [202]. However, at the time of writing, *none of the current Android sandboxes meet the anti-evasion, maintainability, and scalability criteria at once.* Some fully emulated sandboxes (e.g., DroidScope and CopperDroid) cannot offer stealthiness or transparency, leaving several artifacts allowing an app to identify their presence quickly. On top of that, these solutions are also based on obsolete versions of Android. Container-based sandboxes, such as VPBox,

on the other hand, claim to be resilient to evasion by running on actual phones (bare metal). This, however, comes at the expense of scalability because smartphones' hardware still needs to be improved in computing power, allowing only a limited number of analysis containers to run in parallel on the same phone and making container-based sandboxes unfit for large-scale measurements or part of heavy-load analysis pipelines. Finally, all the state-of-the-art solutions are not easily upgradable with the latest Android OS versions, significantly reducing their maintainability.

To fill this gap, this chapter also introduces the design of a new sandbox – named *Droid-Dungeon*– that jointly meets the anti-evasion, scalability, and maintainability requirements and enables us to perform the first analysis of evasive controls in malware and goodware. The sandbox's design allows the deployment in both an actual device and an emulated environment, ensuring, in any case, a high level of transparency.

In short, *DroidDungeon* relies on kernel and user probes to monitor the apps' behavior and return "fake responses" from system calls and framework APIs to hide the underlying emulator and bypass the anti-behavioral controls. While simple on paper, providing fake responses hides several technical challenges, including parsing and modifying Java objects from outside the managed runtime. Moreover, to avoid side effects, *DroidDungeon* has to modify only the events triggered by the app's logic and not those that ensure the correct functioning of the Android operating system. Therefore, it understands *when* to enforce the anti-evasion by performing a custom stack unwind to determine a system or API *call provenance* and precisely determine whether the app under analysis generated a particular event.

Finally, this work leverages *DroidDungeon* to perform the first study of the current usage of evasive checks in modern benign and malicious Android apps. We carefully selected *20,556* malicious and *21,154* apps for our experiments. Malware samples are uniformly spread over 200 families collected by the VirusTotal [228] live feed until April 2023. Conversely, benign apps were retrieved directly from the Google Play store, with a limit of 500 apps for each available category.

This research aims to answer the following main questions.

RQ12: To what purpose does Android malware use anti-behavioral controls?
RQ13: How do malware and goodware differ in using evasive techniques?
RQ14: What is the relation between evasive controls and packers/protectors?
RQ15: Which operations are hidden under evasive controls?

What we discovered is extremely interesting. Malware mainly leverages evasive controls to verify the environment in which they are executed; in particular, almost 70% of evasive

malware aims to detect the emulated environment. We also detected one malicious sample that leverages the SafetyNet [104] Attestation API to verify the legitimacy of the environment. This could be a critical tipping point because comprehensive remediation for SafetyNet (and the new Play Integrity API) does not exist. It also shows how "benign" services can be abused maliciously. Our experiments also show that the evasiveness of malware heavily depends on its family: different malware families implement different numbers and types of evasive techniques.

Benign samples are instead more prone to check app-specific features through APK tampering verification controls to protect themself from client-side attacks. For instance, more than 88% of evasive goodware verifies from where they were installed, and 82% implements at least one IET control related to signature verification. Moreover, while malware uses evasion techniques predominantly at the beginning, goodware often spreads them over its entire execution.

Moreover, we checked if a relationship between evasive controls and packers exists. We discovered that the presence of a packer does not affect the number of evasive checks but rather the sample's techniques. In particular, packer samples are more prone to check process artifacts than non-packed ones.

Finally, we analyzed every sample two times: in the first run, *DroidDungeon* hides the underlying emulator by enforcing the anti-evasion criterion, while in the second, the app is executed and monitored without modifying the emulator's behavior. The results highlight that evasive samples perform different events depending on when they are executed. In particular, 14% of evasive malware samples stop their execution before launching any activity when executed in an emulated environment. Also, evasive malware is more prone to interact with potentially dangerous APIs (e.g., record audio and video of the device) or run CLI commands if executed in an environment that behaves like an actual device.

In summary, this chapter provides the following contributions:

- the collection of all documented evasive techniques and their implementation by categorizing them into two main groups and a proof-of-concept Android app that implements all of them;
- *DroidDungeon*, a novel Android sandbox for dynamic analysis of both benign and malicious apps that fulfills the anti-evasion, scalability, and maintainability requirements at ones;

- a deep analysis of the actual usage of anti-behavioral techniques in modern Android apps – both benign and malicious. Moreover, this study investigates the operations a malware tries to hide under such controls.

6.2 Taxonomy of Android Evasive Controls

This section introduces the malware behavior and method concepts according to the name convention of the MITRE Malware Behavior Catalog [46]. It discusses our categorization of evasive techniques based on how they are implemented.

6.2.1 Behaviors and Methods

Section 2.2.2 categorized anti-behavioral techniques based on their purposes and highlighted three different objectives, namely environment, APK tampering, and high-level verifications. However, there are many ways to achieve the same objective, and each can be implemented differently. Following the MITRE jargon, in this context, a malware *'Behavior'* is the target of a specific control, which can be implemented in multiple ways, called malware *'Methods'*. For instance, root and hook detection are different malware behaviors aiming to achieve the same objective (i.e., environment verification). Moreover, the malware methods (i.e., the implementations) for each behavior could be very varied; for instance, some of the root detection tests aim at verifying the presence of some executable files (e.g., the `su` binary), while other controls check whether well-known paths that are usually read-only have write permission.

From now on, we assign a unique identifier to each evasive method, which consists of the concatenations of three strings: the behavior, the method, and the type of control. For instance, the ROOT-SU-FILE method denotes a *root detection* evasive behavior, which aims to verify the presence of the `su` binary (method) and achieves that by looking at the *file* (type of control). Table C.1 in the Appendix summarizes all the 97 unique methods. Moreover, we also developed Android-Al-Khaser [15], an Android app that implements a proof-of-concept of each evasive behavior.

6.2.2 *Direct* and *Indirect* Evasive Techniques

This part introduces our novel evasive techniques categorization based on the malware methods. In other words, this categorization considers how different malware behaviors are implemented. It extends the concept proposed by Petsas et al. [168], who classified evasive

controls based on the information they rely upon: I) *static heuristics* check static information that is initialized to known values (e.g., a well-known path), II) *dynamic heuristics* observe unexpected behavior of the environment, and III) *hypervisor heuristics*, based on incomplete emulation of the hardware device. It is worth noticing that this categorization is orthogonal to the one proposed in Section 2.2, which is based on the goal of the techniques regardless of how the samples archive it.

It is essential to notice that while, in some instances, we can detect the presence of a given evasive method used by an app, in other cases, an app can collect some general information that can be used internally to implement the evasive control. We call the first type a *Direct* evasion technique (DET) and the second an *Indirect* one (IET).

For instance, Magisk [223] is a famous open-source app that customizes Android, requiring root access. To verify if this app is installed, a developer can interact with the `getPackageInfo` or the `getInstalledApplications` methods of the `PackageManager`. The former accepts the package name of the target app, while the second does not take any argument for filtering the result and returns a list of *all* apps installed for the current user. Thus, if a sample invokes the `getPackageInfo` method with the `com.topjohnwu` argument (i.e., the package name of the Magisk app), we can flag it as a direct implementation of the root detection evasive method. However, a sample can also retrieve the entire list of the installed apps by invoking the `getInstalledApplications` and then look for the Magisk package name in several stealthy ways (e.g., by comparing the hash of each name). Hence, in that latter case, we can not be sure that the app is trying to evade the analysis; however, we can still report that it collects specific system/device information that can be used to implement evasive control in an *indirect* way.

6.3 *DroidDungeon*

Figure 6.1 presents an overview of *DroidDungeon*. The monitoring and anti-evasion capabilities of the sandbox are implemented in a custom kernel module (from now on, *CKM*). This choice is less invasive than other techniques commonly used in malware analysis – such as userspace instrumentation, as pointed out by previous work [156, 211] – resulting in stealthier and sounder analysis. Precisely, *CKM* can monitor system and app *events* (i.e., Android APIs, library functions, and system calls), enforcing the anti-evasion criterion when necessary. Alongside the *CKM*, *DroidDungeon* also ships a userspace companion app (*UCA*), which provides configuration directives to the kernel module through a special device file (`/dev/mydevice` in the figure). These directives include the list of Android APIs and

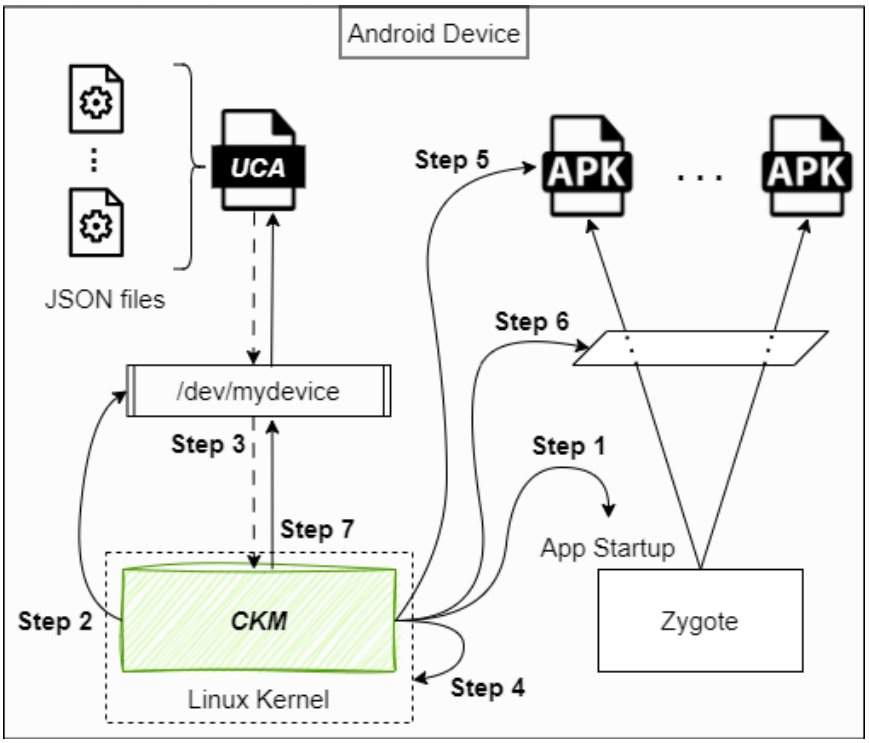


Figure 6.1 Overview of *DroidDungeon*

syscalls to monitor and the anti-evasion techniques to enable, which the analyst can tweak by providing configuration files to the *UCA*.

At first, the kernel module is automatically loaded during the device boot routine but remains dormant, waiting for the *UCA* to start (Step 1 of Figure 6.1). *CKM* detects this event by monitoring the `prctl` syscall that the Android operating system uses to name the main process of an app when it starts. At this point, the kernel module finds the Zygote's PID by iterating over the list of all running processes, and it creates a special device file (Step 2) to communicate with the *UCA* (Step 3).

The app then sends the categories of syscall that the module needs to hook (Step 4) and the Android APIs to monitor (Step 5). To give an idea, the syscall *connect* belongs to the network category. Table 6.1 recaps the list of syscalls for each category.

The kernel module leverages the kernel *tracepoint*, *k(ret)probe*, and *u(ret)probe* subsystems, which are built-in tracing mechanisms provided by the Linux kernel. Tracepoints and K(ret)probes allow one to register callbacks triggered when a specific kernel function is executed. Each callback controls the calling process, including inspecting and modifying CPU registers and memory. While tracepoints can only be declared at compile time, k(ret)probes can be defined at runtime and allow tracing a specific function's beginning and

Table 6.1 *DroidDungeon*– mapping between category and system calls

Syscall category	**System call**
ACCESS_FILE	(f)access(at), open(at), *stat*, readlink(at), getdents64
EXE_COMMAND	execve(at)
PROCESS_MNG	clone, (v)fork, kill, wait(id|4), ptrace, pipe(2), tee, mq_*, sched_*, rt_sigaction, signalfd(64)
FS_MNG	getcwd, (f)chdir, renameat(2), mkdir(at), rmdir, *link*, *chmod*, mknod(at), inotify_add_watch, fanotify_mark, (p)poll, *xattr, flock, mount
NETWORK	socket, binde, accept(4), connect, getsockname, listen, getpeername, sendto, recvfrom, sendmsg, socketpair, setsockopt
SYSTEM	reboot, getcpu, sys*, uname, time*, clock_*
IDENTITY	*gid, *pid, *uid, *sid
MEMORY_MNG	mmap, *mprotect

end. In addition, u(ret)probes are the userspace equivalent to k(ret)probes and can be placed on any ELF or OAT file a process loads. In particular, *CKM* sets one tracepoint to tap the kernel's syscall trap handler that parses the parameters of each syscall invoked on the system and one k(ret)probe or u(ret)probe for each specific syscall and Android API that logs the event and implements the relevant anti-evasion tricks.

During the module initialization phase, the *UCA* also sends the memory addresses of the `android.os.Build` Java class. This class is initialized during the device's start-up in the Zygote process, and some of its static fields are known to be exploited for evasion. Since each app inherits its address space from Zygote in Android, virtually all apps can access such valuable information, which *CKM* needs to modify as an anti-evasion countermeasure (Step 6). In theory, it is possible to find the address of the `Build` class directly from kernel space by introspecting any app's address space and parsing the ART heap, but we opted to offload this task to the userspace app instead. Obtaining the class address is much easier by taking advantage of the managed ART runtime from the app than it is from kernel space.

The module setup phase ends when the *UCA* sends the *init* command. At this point, the kernel module works event-driven, responding to the events triggering the registered tracepoints and probes and sending the corresponding log entries to the companion apps through the special device (Step 7).

6.3.1 *CKM*: Implementation Details, Technical Challenges & Solutions

The *CKM* is a loadable Linux kernel module written in about 11,842 lines of C code. It can intercept all the system calls, kernel functions, and compiled userspace code in ELF or OAT format and parse and modify native (C/C++) and Java objects. *CKM* monitors only the apps installed after the companion one by filtering out all the events of processes with a UID higher than the *UCA*. This prevents *CKM* from interfering with critical system components, which would make the operating system unstable. Last, *CKM* sends a log of all relevant recorded events to the *UCA* through the device file by using a protocol based on the eXternal Data Representation (XDR) standard [124].

Tracepoints and probes. To register tracepoints, kernel, and user probes, the *CKM* uses respectively the `register_trace_sys_enter`, `register_kprobe`, and `uprobe_register` standard Linux kernel functions. While `register_kprobe` and `uprobe_register` can be used by any kernel module, the Linux kernel prevents an external module from registering such a tracepoint. We circumvented this limitation by patching the Android kernel to export tracepoint-related functions and make them externally visible. This tweak consists of only two lines of C code (as shown in Listing 6.1), making it, in fact, extremely portable between the various kernel and Android versions.

Listing 6.1 Android kernel modification to enable system call tracepoint in an external kernel module.

```
1    // register_trace_sys_enter
2    EXPORT_TRACEPOINT_SYMBOL_GPL(sys_enter);
3    // register_trace_sys_exit
4    EXPORT_TRACEPOINT_SYMBOL_GPL(sys_exit);
```

Unlike tracepoints, whose targets are fixed and defined at compilation time, the *CKM* has to specify the targets of the kernel and user probes at runtime. The former can be set by specifying the kernel symbol name of the desired function to hook. The procedure is more complicated for the latter, as the `uprobe_register` function requires a specific offset of the ELF file at which to add the probe. We offloaded the offset calculation to the *UCA*, which delivers the binary file paths (i.e., the path of the ELF or OAT file) and the offset of each API to hook through the communication device.

Parsing probe parameters. To monitor execution events and implement anti-evasion measures, *CKM* needs to interpret the parameters the analyzed apps employ to invoke syscalls and APIs. The first step of this process consists of recovering the parameters' values according to the architecture calling convention. While implementing the parsing routines, we discovered that the OAT files adopt undocumented (at least to the best of our

knowledge) calling conventions on the x86 and x86-64 architectures. In particular, the first three parameters must be passed through registers on x86 or the first five in the case of x86-64, with the remainder on the stack. Moreover, Java instance methods have a hidden parameter pointing to the object on which the method is called (referred to as *this* in Java). Once the parameters are retrieved, *CKM* handles them according to the related event's semantics. For instance, to monitor the write syscall, *CKM* resolves the first parameter to the path of the output file and copies the data written by de-referencing the pointer provided as the second parameter.

Java object parser. While parsing syscall parameters is relatively easy, handling the parameters provided to Android APIs proved to be a more complex task requiring knowledge about how the ART runtime stores Java objects in memory. In particular, each Java entity can be classified into four categories according to its type, namely Primitive, Array, String, and Complex [11]. While Primitive entities store basic data types (e.g., bool, char, int), all the other categories represent Java *objects*, i.e., structured data types that inherit their structure from the `java.lang.Object` class. The Object class has two members: a four bytes pointer to a `java.lang.Class` object and a four bytes hash.

The object's Class defines the size of the object instance, its superclass, and the list of fields as an array of `Art_Field` objects. Each `Art_Field` specifies a pointer to the declaring class of the field (from which the field name can be retrieved) and its offset (in byte) from the beginning of the Complex object. Thus, by parsing the array of `Art_Fields`, we can decode the Complex objects as a set of fundamental object instances. It is also important to note that a field of a Complex object could, in turn, be another Complex object or an Array of Complex objects. This makes parsing Java objects a recursive procedure that ends when a Primitive or a String object is found. To our knowledge, *CKM* is the first in-kernel runtime Java object parser and modifier.

Custom stack unwind. *DroidDungeon* has mainly adopted a "fake response" approach by modifying the output values of several system calls, Android APIs, and library functions when invoked by the app under analysis, mimicking the behavior they would show on an actual device. However, indiscriminately providing fake responses to all invocations of these functions can be detrimental. For instance, several evasion techniques include scouting the file system for QEMU-specific device files. A naive approach to counter these stratagems would be to counterfeit the result of all file system syscalls that could reveal their existence. In doing so, however, we would hinder the intended uses of such devices that the Android framework opens any time a new app starts, resulting in an unrecoverable exception.

To handle this and other cases, we developed a more fine-grained technique to assess a call's purpose by considering its *provenance*. *CKM* tampers its outputs only if the call originates (directly or indirectly) from the app's code. On the other hand, if the call does not originate from the app's code, it must be part of the intended operations performed by the Android framework, and *CKM* should not meddle in its execution.

To assess a call's provenance, *CKM* unwinds the call stack, reconstructs the list of function calls, and looks for those that belong to the app's code. Reconstructing the calling function list means retrieving the list of return addresses from the stack. The first element in this list (i.e., the caller's address of the hooked function) is the value in the stack immediately before the area pointed by the stack pointer. The following elements in the list can be computed iteratively by reconstructing each caller's stack frame and retrieving the respective return address.

At any given step of the stack unwind procedure, *CKM* computes the next frame pointer (FP) as:

$$FP_{(n+1)} = FP_n - FS_n - (sizeof(void*) * SA_n)$$

where FS_n is the frame size of the n-th function in the call stack (i.e., the sum of the sizes of all stack variables of that function), and SA_n is the number of its parameters passed on the stack. Since neither FS nor SA can be easily retrieved at runtime, we opted for computing them *a-priori* for every symbol of every library in the Android framework. To this end, we developed a Ghidra [79] script and a Python program based on *oatdump* that computes each function's frame size and stack parameters for Android ELF libraries and OAT files, respectively. The value calculated for each function in the Android libraries is then embedded in the *CKM* at build time.

The stack unwind procedure ends whenever it encounters an address that belongs to the app's code or an invalid address. In the first case, the module verifies which method or API the app invokes to determine whether to enforce the anti-evasion policy. When, instead, the unwinding procedure reaches an invalid address, *CKM* infers that the event was due to an Android standard routine (e.g., app start-up) and does not enforce the anti-evasion criterion.

Notice that this approach works even against reflection. Invoking a framework method through reflection only adds a few function calls between the caller and the callee, which our stack unwind routine can handle seamlessly.

To the best of our knowledge, *DroidDungeon* is the first analysis system that performs *call provenance* test through an in-kernel stack unwinder.

JIT & stack unwind. In Android, the `libart.so` library enforces the JIT compilation and contains the *jit functions* to manage all DEX instructions (i.e., `nterp_op_<inst_name>` functions). For instance, `nterp_op_return_void` and `nterp_op_return_object`, respectively, handle the return statement of a Java (`void`) procedure or a function that returns an `object`. Android interprets the non-OAT-compiled DEX code to build a chain of equivalent jit functions to execute.

We discovered that also these functions do not respect the standard calling convention: the jit functions do not have a prologue and epilogue, ending the procedure with a `JMP` instruction instead. Moreover, the return address – a pointer to the following jit function in the chain – is computed at runtime. Thus, the stack unwinding routine does not apply to this scenario. The only exceptions are the `nterp_op_return*` jit functions, which have an epilogue and end with the standard `RET` statement returning to the pointer stored on the stack.

The idea we had to manage JIT-compiled functions in the stack unwind procedure is that once the execution reaches the jit function chain, the stack frame does not changes until the return is reached. It is worth noticing that a JIT-compiled function always ends with a return statement regardless of the first DEX instruction. Also, during the stack unwinding procedure, *CKM* reaches a JIT-compiled function if and only if it invoked another API or a native method, i.e., if it came from one of the `nterp_op_invoke_*` jit functions. Thus, once one of these functions is reached, *CKM* can consider as if the return jit function had been invoked: it adds the return's stack frame size to the SP, retrieving the caller's return address from the stack as usual.

6.3.2 Userspace Companion App

The *UCA* is a regular Android app written in about 1,620 lines of Java and 1,283 lines of C code. It supports the *CKM* in Steps 2 and 7 of Figure 6.1. The JSON configuration files specify the set of functions the *CKM* has to hook and which anti-evasion techniques need to be enforced. Contrary to the kernel probe, which can be registered by specifying only the function name, *UCA* retrieves the offset of the target userspace functions from the ELF or OAT files, which *CKM* needs to register the uprobes. In particular, the app leverages the `readelf` [125] utility for the ELF files, while it uses `oatdump` for the OATs. Finally, *UCA* listens on the device file to log the intercepted events into a regular file.

6.3.3 Anti-evasive Policy

The *DroidDungeon* design allowed us to limit the artifacts we need to manipulate to only the emulator and network categories (refer to Section 6.2). Specifically, once the kernel module receives the init command, it modifies the properties of the network interfaces (e.g., their names) to make them appear similar to the ones in an actual device. Also, it installs probes to monitor and tamper with file-related operations (Step 4 of Figure 6.1) and manipulate high-level Android APIs (Step 5), including changes to system properties (e.g., `ro.hardware`) and the simulation of real sensors.

We recall that some static fields of the `Build` class can be exploited for evasion, and *CKM* modifies them each time a new app starts (Step 6).

Finally, the device has a user logged in to an actual Google account, and the file system is populated with a collection of documents, images, and other common files to resemble a legitimate smartphone so that malware may identify it as a more valuable target [150].

6.4 Experimental Setup

This section provides technical information on how the experimental campaign was carried out. It presents the dataset and then introduces the environment in which we executed the samples.

6.4.1 Dataset

To perform our analysis, we built a comprehensive dataset of Android apps divided into malware and goodware samples. We collected malicious apps from the VirusTotal (VT) feed [228], a real-time stream of JSON-encoded reports containing the analysis results for each app submitted to VT. We wanted our dataset to be diverse regarding the number of families and balanced so that every malware family is well-represented. For this reason, we monitored the feed from September 2022 to April 2023. We only retained the Android apps identified as malicious by at least five engines and fed them to the AVClass2 malware labeling tool [192], which outputs the most likely family name for the sample. At the end of the collection, we ended up with *20,556* malicious apps, uniformly distributed over 200 families. Since our goal was to collect 100 “running” samples for each family, we gathered, on average, 110 for each of them to be able to replace any broken APK that we could not run in our sandbox.

For goodware, we collected the package names of the 500 most downloaded free apps for each of the 50 official Google Play Store categories in April 2023. We extracted this information using Google Play Scraper [61] and downloaded the apps thanks to apkeep [56] directly from the Google Play store. In total, we collected *21,154* unique samples.

6.4.2 Runtime Environment

We used *DroidDungeon* to conduct the first analysis of public evasion techniques in goodware and malware; thus, we implemented it in an Android emulator. Since February 2020, Google has introduced support for running ARM binaries on x86 (Android 9) and x86-64 (Android 11) system images. However, starting from version 12, Android emulators can execute only 64-bit binaries. Thus, to execute ARM and Intel 32- and 64-bit-based Android apps, our analysis system comprises two emulators based on Android versions 12 (API level 31) and 11 (API level 30). Before running an app, we check if it only has 32-bit libraries and choose the appropriate emulator.

Moreover, because *DroidDungeon* can not hook DEX code, we must ensure that every Android Java function is OAT compiled. Thus, we replaced all the OAT framework files in our Android emulators' official `system.img`. We exploited the AOSP build tools to unpack the image, OAT-compile all the Java APIs, and repack it in a new one. It is worth noticing that this step is not straightforward: Android put in place several techniques to validate the integrity of the framework files; thus, we took into account every check. For instance, since version 8, Android performs the verified boot process [105], which assures the integrity of the framework software, also verifying the `system.img`. Thus, we had to recreate a valid `vbmeta.img` file, namely, the data structure contains all the metadata for the verified boot process. In this way, we deployed *DroidDungeon* with the complete support of the Google Play services and apps.

All emulators run on a dual-core 2.10 GHz x86-64 processor, 2 GB of RAM, 16 GB of internal storage, and an 8 GB emulated SD Card. Finally, we manually tested twelve apps (both malicious and benign) that we knew how they worked and found no user experience problems or malfunctions.

6.4.3 Red and Blue runs

The concept of red pill and blue pill in evasive malware [167, 55] refers to the movie "The Matrix", where the protagonist is offered a choice between a red pill that reveals the true nature of reality and a blue pill that keeps him in a simulated world. Thus, in our scenario, a

red pill is an evasive technique, while its blue pill is a corresponding defensive technique to counteract the red one.

In this work, we also wanted to study how much the executions vary when *DroidDungeon* gives blue pills compared to when it does not provide them. Hence, every app in our dataset is executed twice. In the first run (*BlueRun*), *DroidDungeon* hides the underlying emulator by administering blue pills (i.e., it enforces the anti-evasion criterion). In contrast, in the second run (*RedRun*), the app is executed without modifying the emulator's behavior.

In every app execution, *DroidDungeon* stimulates the app user interface for 4 mins to increase its code coverage and simulate a real user with ARES [181]. This black-box tool uses Deep Reinforcement Learning to test and explore Android apps. Moreover, we modified ARES to perform the same sequence of user clicks for every tested app to compare the two execution traces.

Preliminary results. We could correctly execute the 93% of malware (19,090/*20,556*) and the 99% of goodware (21,081/*21,154*). For the failed apps, we were not able to install them because the signature was not valid or the APK file itself was corrupted.

6.5 Results of the measurement

This section discusses the results of our analysis of the malware and the goodware datasets. Starting from the app's execution traces, we developed a post-analysis routine that identifies DETs and IETs by looking at the list of events that occurred after the first access to the `base.apk` file (which, as explained in Section 5.5, signals the start-up of an Android app). When reporting the results of evasive behaviors and methods, we will use the unique identifier (in uppercase) we introduced in Section 6.2 and reported in Table C.1.

6.5.1 Prevalence

DETs and IETs usage. 90.8% of goodware and 68.5% of malware implement at least one DET evasive technique, while about 90% in both categories contain at least one IET. On average, the malware uses 2.1 unique DETs ($\sigma = 1.9$) and 12.8 ($\sigma = 6.7$) IETs, while goodware 3.4 ($\sigma = 2.2$) and 14.4 ($\sigma = 4.9$), respectively. Interestingly, goodware has almost twice as many DET controls as malicious samples on average. However, the sample that employs the maximum number of unique DETs (15) and IETs (39) controls is malicious, which is almost 15% higher than the maximum for goodware.

While the prevalence is high in both groups, there are essential differences in the techniques adopted by malware and goodware. Table 6.2 reports the three DETs and IETs that differ the most among the two groups. In particular, it shows the percentage of malware and goodware that implement a specific technique w.r.t. all the dataset apps. A negative value in the rightmost column means that such an evasive check is implemented more often in goodware, while a positive means is more prevalent in the malware dataset.

Table 6.2 Malware vs Goodware: Differences in adopting evasive techniques.

	Evasive Technique	**Malware**	**Goodware**	**Malware - Goodware**
DET	INSTALL-SOURCE-API	43.3%	88.7%	–45.4%
	VIRT-UND_PERMS-API	35.9%	76.3%	–40.4%
	ROOT-SU-FILE	18.3%	56.1%	–37.8%
	EMU-SYSTEM-API	44.1%	19.8%	+24.3%
	EMU-QEMU-FILE	3.9%	1.2%	+2.7%
	EMU-KNOWN_EMU-FILE	3.2%	1.1%	+2.1%
IET	SIGNATURE-APP-APP_INFO	25.6%	76.2%	–50.6%
	NET-SSL_PINNING-API	54.6%	78.1%	–23.5%
	VIRT-FAKE_COMP-API	31.2%	47.1%	–15.9%
	HOOK-PROC_ART-MAPS	26.2%	13.0%	+13.2%
	HOOK/ROOT-APPS-INST_APPS	19.4%	6.5%	+12.9%
	NET-INTERFACE-NF	14.7%	5.9%	+8.8%

First, more than 88% of goodware verifies how the app was installed or updated by invoking the `getInstallSourceInfo` method of the `PackageManager` (`INSTALL-SOURCE-API`), clearly showing how developers care if the app comes from the expected store.

Interestingly, 76.3% of goodware verifies the environment in which they are executed by checking if permissions not declared on the Manifest are granted to the app (`VIRT-UND PERMS-API`). It is a common technique to detect whether the app is running in an app-level virtual environment [139, 214] because container apps have to declare all possible permissions to manage with a generic plugin app. However, we have investigated this further, discovering that most goodware samples import third-party libraries for analytics and monetization (e.g., [19, 37, 67, 114]), which check the granted permissions at runtime to extract as much information as possible.

The third DET control is related to root detection: 56.1% of benign apps check the presence of the `su` file (`ROOT-SU-FILE`).

On the other hand, in malware, DETs controls are related to detecting the emulated environment or an analysis system. In particular, 44% of malware retrieves and checks

Android system properties, such as the device or the subscriber ID, by interacting with Android managers (`EMU-SYSTEM-API`). In addition, malware samples are more prone to searches for well-known emulator artifacts (e.g., `EMU-QEMU-FILE` and `EMU-KNOWN_EMU-FILE`) and network proxy apps.

Concerning the IETs, goodware often retrieves information about the certificate used to sign the APK by querying the package manager (`SIGNATURE-APP-APP_INFO`), performs SSL pinning (`NET-SSL_PINNING-API`), and checks for artifacts in the process components through the Android APIs (`VIRT-FAKE_COMP-API`).

Conversely, the main IET controls in malware are related to anti-hooking, root checks, and network artifact detection. First, malware verifies process artifacts by checking the content of the `maps` file in the proc file system (`HOOK-PROC_ART-MAPS`). Moreover, almost 20% retrieves the list of all installed apps on the device and then performs some checks over it (i.e., `HOOK/ROOT-APPS-INST_APPS`). This technique can be exploited for detecting hooking (e.g., Xposed) and rooting (e.g., Magisk) apps. Even if there is proper permission for doing it, in Section 5.7, we demonstrated that there are tricks a malicious app can leverage to bypass this protection mechanism. For instance, restrictions are not applied to apps targeting API level 30 or lower, which can retrieve the metadata of any app in the system. Interestingly, 90% of malware on average targets an older API ($<=$ 30), and almost 3% requests the QUERY_ALL_PACKAGES permission. The picture is different for goodware: only 14% targets an API before level 30. Compared to malware, it is a small value, but, in absolute terms, it is unusual that benign goodware apps do not update the target API as guidelines.

Last, malware verifies network interface properties by using native functions, such as `getifaddrs` (`NET-INTERFACE-NF`), to figure out if the device uses a VPN.

SafetyNet & Integrity APIs. During our analysis, we also investigated if and how goodware and malware use one of the SafetyNet APIs or the Integrity API (introduced in Section 2.2.2). Since these technologies are not open-source, we manually analyzed the Android framework to determine their inner workings.

For SafetyNet, we intercept its binder request, the typical Android inter-process communication, and the remote method invocation technique. Table 6.3 reports the entire mapping between binder methods id of the SafetyNet client to its "high-level" security check. Both goodware (35.7%) and malware (30.4%) interact with at least one of the security services SafetyNet offers.

One of the services offered by SafetyNet is Verify Apps. This service is unavailable by default, but an app can ask users to activate it through the `enableVerifyApps` method. Then, an app can check if it has been enabled using the `isVerifyAppsEnabled` method. Once

Table 6.3 Mapping between SafetyNet Binder and high-level security mechanisms.

Id	High-level check	Retrived data
7	Attestation API	`AttestationResponse`
4 14	Verify Apps API	`VerifyAppsUserResponse`
5 13	Verify Apps API	`HarmfulAppsResponse`
12	Safe Browsing API	`InitSafeBrowsingResponse`
3	Safe Browsing API	`SafeBrowsingResponse`
6	reCAPTCHA API	`RecaptchaTokenResponse`

the service is enabled, users can check the list of harmful apps by invoking the `listHarmfulApps` method. Among samples that use SafetyNet, over 95% of malware and 90.6% of goodware samples communicate with the Verify Apps service, and all invoke the `isVerifyAppsEnabled`. However, surprisingly, just one goodware invokes the `listHarmfulApps` method.

Also, about one-fifth of malware (14%) and goodware (19%) use the Safe Browsing API: they invoke the `loadUri` method to check whether a URI is linked to a well-known threat. However, this measurement is limited: we cannot distinguish whether it is an explicit invocation because the Android WebView automatically invokes this mechanism. Starting in April 2018, WebView supports the Safe Browsing feature by default [106, 82], automatically verifying the URI through this method.

Regarding the Attestation API, we observed only 0.8% of goodware samples and just *one* malware that leveraged it to verify the legitimacy of the execution environment. It is worth noticing that, contrary to other SafetyNet services, the Attestation API requires an app to have a registered and valid API key on its Google website; however, this malicious sample demonstrates that the attackers can abuse "benign" security mechanisms. Moreover, our data confirms the findings of Ibrahim et al. [113], showing that legitimate apps do not adequately use SafetyNet services that would significantly improve their security posture.

We intercept the intents used to communicate with this component to monitor the Play Integrity. Given that this service is very recent (explicitly created to replace SafetyNet), we monitored only a negligible amount of goodware (0.11%) and no malware using this API. This new service will be harder to exploit by malicious actors because it is strictly tied to Google Play Services and requires many verification steps.

Time-based Analysis. We normalized the execution time of each sample in a [0,100] range (as suggested by [143]), and then divided it into three time slots: [0-10], [11-89], and [90-

100]. Then, we tracked when the first and last evasive checks (both DETs and IETs) were performed. Moreover, we also considered the difference between the last and the first to examine whether checks are usually executed in quick succession or at different points in time. From a preliminary analysis, we noticed that there were no significant differences if we considered DET and IET separately; therefore, for ease of reading, we consider them together.

The KDE plots for malware and goodware are shown in Figure 6.2. The percentages of the first and last evasion techniques occurring during the first slot [0-10] of the execution are 63.6% and 10.2% for malware and 73.4% and 2.7% for goodware, respectively. Interestingly, most goodware performs evasive checks at the beginning of execution, even more than malware. Then, the last evasive technique falls in the last slot [90-100] for the 30% of malware and 39.3% of goodware.

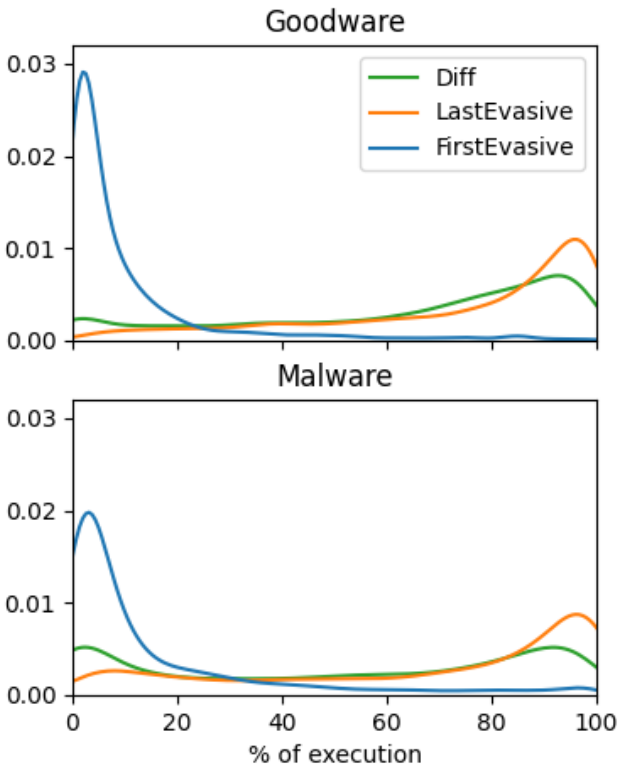


Figure 6.2 Kernel Density Estimation for the timing of DET and IET controls in malware and goodware.

Then, for each slot, we computed the percentage of how many times a specific control has been used. For goodware, the three most common evasive controls are `INSTALL-SOURCE-`

API, ROOT-SU-FILE, and VIRT-UND_PEMRS-API, regardless of the time slot. Thus, benign samples verify from where they were installed and whether the su binary file is present or some non-declared permission is granted. On the contrary, for malware, evasion checks depend on the timing. In the first time slot, malicious samples verify the maps file in the proc file system (HOOK-PROC_ART-MAPS) and from where they are installed (INSTALL-SOURCE-API). Then, regardless of the second and third slot, they look for Android emulator fingerprints (e.g., device ID) and network-related information (NET-INTERFACE-API, i.e., the name of the network interface).

Finally, we also investigated the order in which goodware and malware samples perform the evasive checks. Evasive goodware is predominantly characterized (54%) by controlling the installation source (INSTALL-SOURCE-API), while only 6% of evasive malware uses it as the first control. On the other hand, for about half of the malicious evasive samples, the first control is HOOK-PROC_ART-MAPS, followed by EMU-SYSTEM-API (12.5%) and NET-LISTENER-API (10%). The order in which apps utilize different strategies is crucial because researchers need to appropriately mitigate them to avoid limiting the results to those evasive techniques used first.

Evasive among families/categories. We assessed if the goodware category (e.g., banking or game apps) or malware family affects the overall evasiveness of a sample. For instance, it is reasonable to assume that benign samples are more prone to implement evasive techniques if they manage sensitive user information (e.g., banking and finance).

We found that 195/200 (97.5%) malware families contain at least one sample that uses a DET, while if we include the IETs, the number of families goes up to 199/200. Also, for 22/200 (11%) malware families, *all* the samples in the family contain at least one DET. These numbers indicate that it is crucial to consider this phenomenon in the dynamic analysis of Android malware. On the other hand, all goodware categories contain at least one evasive sample, but none have all samples with at least one DET technique. Moreover, we measured the variation of the number of evasive techniques in malware w.r.t. goodware. On average, the standard deviation of malware is almost four times w.r.t. goodware; namely, the number of controls carried out by malware apps depends on their family. In contrast, the app category only affects a small number of evasive checks in goodware.

We also assessed the 'evasiveness' of a family by counting the number of evasive techniques for each sample in the family. On average, the most evasive malware family is *loead*, whose samples implement, on average, 16.3 evasive controls. This is followed by *fydad* (14.7), *beitaad* (11.2), and *snaptube* (11.1). In the benign dataset, *Finance*, *Entertainment*

(e.g., Netflix and Twitch apps), and *Shopping* are the most evasive goodware categories (with an average of more than 6 evasive controls per sample).

Finally, we measured how many categories/families use a specific evasive technique. For goodware, 21 DETs and IETs are implemented by more than 80% of the app categories, while this number decreases to 12 for malware families. The most widespread for goodware (almost all goodware categories) are verifying process artifacts, checking the APK signature, or whether permissions not declared in the Manifest are granted. On the other hand, the most common for malware are emulator (e.g., `EMU-SYSTEM-PROPS`) and hook detection checks (e.g., `HOOK-FRIDA-FILE`). Interestingly, some checks (e.g., `EMU-KNOWN_EMU-FILE` and `HOOK-PROC_ART-MAPS`) are more spread in goodware categories (86%) w.r.t. malware families (32%), even if their overall occurrence is higher in malicious apps (e.g., `EMU-KNOWN_EMU-FILE` occurs in the 4.3% of malware and only 1.6% of goodware).

RQ12 & RQ13. Our experiments show that goodware and malware use evasive techniques for different purposes. For instance, about 70% of evasive malware implement at least one environment verification control related to the emulator detection, while only one-third of evasive goodware does it. Contrarily, the most common environment verifications for goodware are app-level virtual environment (81%) and root checks (61%), but they account for respectively only 43.4% and 33% of malicious samples. Goodware is also more prone to implement APK tampering verification; e.g., more than 82% of all benign samples perform signature verification controls.

We also observed that malware tends to rely on IETs techniques more often than on DETs. For root detection, goodware verifies if the `su` binary exists, while malware interacts with the Package Manager to retrieve the list of all installed apps.

Finally, our results highlight how malware families affect the type of evasive controls, while goodware apps tend to employ the same evasive techniques regardless of their categories. However, benign apps implement a heterogeneous set of controls for each category because the variation of the evasive controls in every category is higher than in the malware family. Each category has samples with several and no evasive checks, showing how evasive controls are implemented especially by only the most popular apps.

6.5.2 Evasive w.r.t. Packers & Protectors

This part looks at the relationship between evasive behaviors and the presence of packing schemes and software protectors. We used APKiD [178] to determine whether a sample uses packing or protection techniques. On the goodware dataset, APKiD could not recognize any

packer or protector; conversely, it identifies 24 different packers in 11.4% of the malware samples. As reported in Table 6.4, the Jiagu packer is the most common in our dataset and was detected on 43.0% of the packed samples. Tencent, Baidu, and MultidexPacker are the following most prevalent packers, accounting respectively for 9.9%, 5.8%, and 5.3%; the remaining 10/24 packers account for less than 1%. On the other hand, only 0.2% of the malware apps are protected by two different protectors: Virbox and CNProtect. Given the negligible number of protected apps, our analysis is focused only on packers with a non-negligible prevalence ($> 1\%$).

Table 6.4 Best correlation between evasive techniques and packers.

Packer	**% packed apps**	**# evasive**	**Most used evasive**	
APKProtect	5.2%	43	EMU-SYSTEM-PROPS	92%
ApkEncryptor	4.0%	55	SIGNATURE-ZIP-FILE	87%
Baidu	5.8%	37	HOOK-PROC_ART-MAPS	87%
Bangcle	3.3%	9	HOOK-FRIDA-FILE	65%
Bangcle (SecShell)	1.6%	12	HOOK-FRIDA-FILE	84%
DexProtector	4.1%	51	HOOK-STACKTRACE-API	88%
DexProtector for AIDE	1.2%	48	NET-SSL_PINNING-API	88%
Ijiami	4.3%	27	SIGNATURE-ZIP-FILE	90%
Jiagu	43.0%	56	HOOK-FRIDA-FILE	69%
Mobile Tencent Protect	9.9%	43	HOOK-FRIDA-FILE	88%
MultidexPacker	5.3%	26	SIGNATURE-ZIP-FILE	100%
SecNeo.A	1.9%	24	HOOK-PROC_ART-MAPS	58%
Tencent's Legu	2.3%	29	SIGNATURE-ZIP-FILE	93%
Unicom SDK Loader	5.1%	46	HOOK-STACKTRACE-API	50%

In this context, our goal is to understand whether any difference exists in the adoption of evasive behaviors between packed and non-packed samples. We started by measuring the proportion of apps that implement at least an evasive check, and found a similar prevalence between the two classes - respectively 82% and 89% for packed and non-packed apps.

We further investigated whether specific evasive controls are more characteristic of the packed apps than non-packed ones. We found that the most used evasive techniques in packed malware are `SIGNATURE-ZIP-FILE`, `HOOK-PROC_ART-MAPS`, and `HOOK-FRIDA-FILE`. On average, the former IET technique (i.e., opening the `base.apk` file through the `java.util.zip.ZipFile` Java utility) is used by more than 65% of packed apps, with a peak of more than 90% for Ijiami and ApkEncryptor. On the contrary, such a technique is used by 26% of the samples when considering the collection of non-packed malware and less than 20% for goodware. From an evasive point of view, this operation is helpful to access and read specific files in the APK to perform signature checks or code integrity. However,

packers also leverage this mechanism to unpack the APK file (without unzipping inside the disk) and load resources or encrypted code.

Nine packers cause the apps to open the `fd/*`, `task/*`, or `maps` files under the proc file system (i.e., `HOOK-PROC_ART-MAPS`, `HOOK-FRIDA-FILE`) more frequent compared to non-packed ones. In particular, these operations occurred more than 60% of packed apps (with a maximum of 81% for Tencent, Baidu, and Bancled), while this value decreases to 25% for non-packed malware. These IET techniques verify the process artifacts to identify changes and hook mechanisms, such as the instrumentations injected by Frida.

We finally investigated how varied the evasive controls are for each packing software. We detected that APKs packed with Jiagu implement the highest number of unique evasive controls (56 over 97 techniques identified in this study [15]), closely followed by the apps packed with ApkEncryptor and DexProtector, on which we respectively identified the presence of 55 and 51 different evasive techniques. Conversely, we could only measure 9 distinct evasive checks on samples packed with Bangcle. In a deeper investigation, we found that for some packers `all` the samples implement specific evasive controls: `SIGNATURE-ZIP-FILE` always characterizes samples packed with MultidexPacker, while those packed with AppSealing and ChronClickers always control for `EMU-SYSTEM-API` and `HOOK-PROC_ART-MAPS`. Overall, we measured that although some techniques are more prevalent in packed samples or specific packing schemes, all evasive behaviors used in packed samples also exist in non-packed ones and vice versa. This highlights that, in the considered dataset, evasive techniques are not strictly related to the usage of a packer.

RQ14. A comparable ratio (82% and 89%) of packed and non-packed samples implements at least an evasive technique. Similarly, on average, packed and non-packed APKs implement 5.1 and 8.7 evasive checks, respectively. Nevertheless, some behaviors such as `SIGNATURE-ZIP-FILE, HOOK-PROC_ART-MAPS, and HOOK-FRIDA-FILE` are more widespread in packed samples than in non-packed ones. Moreover, samples packed with specific packers always implement a subset of evasive controls: for instance, samples from AppSealing and ChronClickers always check Android system properties (`EMU-SYSTEM-API`) and process memory artifacts (`HOOK-PROC_ART-MAPS`), which are observed on less than 40% of non-packed samples. However, we could not find evasive techniques exclusively employed by packed samples or specific packing routines.

6.5.3 *BlueRun* vs. *RedRun*

In our final experiment, we measured the difference between the two execution traces: the *BlueRun* in which our sandbox mitigates the evasion mechanisms, and the *RedRun* in which it does not. The expectation is that an app employs evasive checks to *avoid* executing a particular piece of code. However, it is not trivial to establish a methodology to compare two traces and find this difference. The reason is that we cannot make assumptions about code that does and does not execute. Other works [231, 202, 4] have faced a similar problem; however, they addressed it for their specific use case: [202] only compared the number of file system operations, while [231, 4] examined different forms of a call graph.

Therefore, we decided to measure the differences between the events in the execution traces by dividing them into 11 high-level categories related to the workings of Android: 1) *Accessibility Service* (a11y), 2) *Broadcast Receivers* (BR), 3) *Command Line Interface commands* (CLI), 4) *Content Providers* (CP), 5) *Dangerous APIs* (DAPI), 6) *Dynamic Code Loading* (DCL), 7) *File System* (FS), 8) *Inter-Process Communication* (IPC), 9) *Network* (NET), 10) *Requests for permissions* (PERM), and 11) *Systems Services* (SS).

To conduct this measurement correctly and verify whether a pair of events were the same, we had to post-process the traces to remove execution-specific values (e.g., a path with UID or temporary network tokens). We were guided by some works [18, 51] where, for example, the authors generated ML features related to file system activity by removing the file name and just considering the path plus the file extension; or else, when dealing with a URL, they were considering only protocol, domain, and port.

In addition, there are some important factors that we considered. First, there may be cases where the sample does evasive checks, but we do not provide blue pills because our environment does not need them. For example, the `su` executable is not present in *DroidDungeon*, so if an app checks for the presence of this file, it will not find it in both executions. On the other hand, if an app checks for the `/dev/goldfish_pipe` file (the presence of this device reveals an emulator) in *RedRun* it finds it, while in *BlueRun* it does not. For this reason, the numbers we will report below must be considered a lower bound.

Second, we considered the events of the two traces divided into categories as elements of two ordered sets that we will call Red (R) and Blue (B) for brevity. $B_{<cat>}$ denotes the set of events in a specific category (e.g., B_{NET} is the collection of NETwork events of *BlueRun*). Of these sets, we calculated union, intersection, the two differences, the various cardinalities, Jaccard Index, and finally aggregated by category, checking in percentages what the significant differences were. We also computed the percentage number of times an

event occurred in B and not R and vice versa. Table 6.5 summarizes these results. In the following, we only report the significant differences for evasive apps between B and R traces.

Table 6.5 Stats about trace comparison in evasive samples, all numbers are in percentage [%]

	Malware					**Goodware**				
	$R = B$	$R \subset B$	$\|B \setminus R\| > \|R \setminus B\|$	$\|B\| > 0 \wedge \|R\| = 0$	$avg \frac{\|R \setminus B\|}{\|B \setminus R\|}$	$R = B$	$R \subset B$	$\|B \setminus R\| > \|R \setminus B\|$	$\|B\| > 0 \wedge \|R\| = 0$	$avg \frac{\|R \setminus B\|}{\|B \setminus R\|}$
a11y	68.01%	10.70%	1.62%	0.00%	62.98	89.27%	4.43%	0.42%	0.00%	61.08
BR	85.90%	9.37%	0.65%	0.14%	93.84	92.38%	4.31%	0.00%	2.94%	0.00
CLI	90.78%	3.65%	1.82%	0.30%	97.22	98.19%	0.00%	1.20%	0.00%	100.0
CP	79.79%	13.28%	1.08%	0.03%	72.14	88.68%	6.23%	0.21%	0.21%	90.00
DAPI	73.60%	9.97%	6.96%	0.37%	83.01	95.56%	1.50%	0.06%	0.17%	100.0
DCL	86.92%	1.88%	0.21%	0.05%	87.50	80.22%	0.86%	0.00%	0.24%	0.00
FS	23.21%	10.47%	34.78%	0.00%	61.43	26.63%	5.69%	28.19%	0.00%	73.46
IPC	74.58%	10.77%	2.96%	0.25%	83.11	78.87%	9.73%	0.56%	0.45%	91.70
NET	48.50%	6.03%	31.48%	0.46%	87.45	29.74%	2.85%	52.45%	0.86%	93.93
PERM	80.37%	10.27%	1.68%	0.00%	83.72	80.01%	4.44%	1.92%	0.00%	90.74
SS	67.59%	25.35%	1.32%	0.03%	84.72	80.15%	4.54%	0.18%	0.00%	100.0

Moreover, although these measurements are only meaningful for the evasive samples, we also verified the non-evasive ones to double-check that our sandbox worked properly. Therefore, for all the numbers we reported, we have verified that the same trend does not occur for the non-evasive samples. During this inspection, we found that irrespective of the classes (malware/goodware) and category except for NET, for the non-evasive samples, the two traces are equal ($R_{<cat>} = B_{<cat>}$) in more than 90% of the samples, while this percentage varies considerably for evasive ones (between 30% and 96%). The only notable exceptions are the network traffic traces: they are very different even in non-evasive goodware samples ($R_{NET} = B_{NET}$ only in 46%). The main reason is related to ads and monetization libraries: at each execution, goodware renders different ads, which generates different network requests.

A closer look at the cases where $R_{SS} \subset B_{SS}$, i.e., when the two traces are not the same and the *RedRun* events are a subset of the *BlueRun* ones, allowed us to observe that almost 14% of malware does not create any Android Activity when they are executed in *RedRun*, but they do it when the anti-evasion criterion is enforced; this phenomenon occurs for less than 4% of goodware. In practical terms, these samples did not show any GUI and finished execution when their evasive controls detected a potential analysis environment. Nevertheless, contrary to other systems (e.g., Windows [55, 143]), most evasive malware samples (86%) do not stop their executions if the anti-evasion criterion is not enforced, behaving like a legitimate Android app.

Then, combining some observations in the DAPI, CP, and SS categories, we noticed two interesting behaviors that evasive malware exhibits when it thinks it is not under analysis. First, almost 10% more samples interact with the Captioning Manager (that contains methods to access and monitor preferred video captioning state), and most of them (70%) use it as an alternative way to get the user's properties, such as the preferred language. Second, 2% records the audio or video of the device for an unlimited period through methods of the `MediaRecorder` class.

Regarding the FS category, more than 17% of evasive goodware looks for the `su` binary in *BlueRun*, while they did not do it in *RedRun*. We observed that these goodware samples perform root checks only after verifying the presence of an emulator. Conversely, it happened for less than 3% of evasive malware, which, on the other hand, are more prone to change the mode bits of files in their private folders. For instance, the `fchmod` syscall occurred more than 20% of evasive malware in the B_{FS} trace w.r.t. the R_{FS} one. We manually investigated this latter fact and found that, in these cases, the malware makes some files containing code, such as native libraries, writable to modify their content (e.g., the Grifthorse Trojan [251]). In this way, the sample will execute a different code at runtime compared to the statically available one in the APK.

Finally, the CLI category is abused by 1.5% of evasive malware to retrieve system properties value through the `getprop` command or run `chmod` command (< 0.1 for goodware), avoiding interacting with the Android APIs.

RQ15. Our experiments show that 14% of malicious and 4% of benign samples refrain from running in our analysis environment when we do not mitigate evasive controls. In other cases, malware hides techniques to obtain information about the device or records audio/video without the user's knowledge. It also overwrites portions of its code to perform different operations than those that could be observed by statically analyzing the APK file. Goodware, instead, tries to hide its search for the presence of a device with root permissions.

6.6 Related Work on Android Sandboxes

According to [202], we grouped current Android sandboxes for malware analysis based on the technique on which they are based. First, this section discusses emulator-based sandboxes and then introduces container-based solutions. There are currently no Android malware sandboxes based on app-level virtualization, so this solution was not considered. Finally, this section discusses the differences between *DroidDungeon* and the other state-of-the-art sandboxes.

Full-system emulation. In the last years, researchers proposed several sandboxes based on Android emulators. In 2012, Yan et al. developed DroidScope [243], a virtual machine introspect (VMI) system to monitor the activity of the malicious app in the Android emulator. DroidScope leverages a 2-level VMI to gather information about the system and exposes hooks to a set of APIs. In 2015, Tam et al. proposed CopperDroid [211], a sandbox built on QEMU to automatically perform out-of-the-box VMI-based dynamic analysis and reconstruct Android malware behaviors. In the same year, DroidBox [170] and CuckooDroid [137] have been proposed. The former is a custom Android OS version 4.1.1 variant that tracks and taints API calls. Alternatively, CuckooDroid is an extension of Cuckoo Sandbox [69] for automating the analysis of Android apps; it is based on the Xposed Framework to monitor API calls and provide blue pills to the target apps. Similarly to CuckooDroid, over the years, other researchers proposed hook-based sandboxes which rely on Xposed [49, 32, 74] or Frida [62]. In 2018, Liu et al. [134] proposed RealDrois, an emulator-based analysis system built by modifying the Android framework.

Android Container-Based Virtualization. Similarly to the desktop counterparts, the Android container-based virtualization is a lightweight in-kernel virtualization technique that creates an isolated (virtual) environment in the same Android device. However, Android container development has to overcome several challenges; in particular, mobile devices are not designed for multiplexing hardware components (e.g., WiFi and Bluetooth).

In 2011, Andrus et al. proposed Cells [14], a virtualization architecture enabling multiple isolated virtual phones (VP) to run simultaneously on the same physical device. In 2015, Xu et al. developed Condroid [240], a lightweight Android virtualization solution based on container technology, which leverages both Linux namespaces and cgroups to create multiple VPs. In 2021, Song et al. proposed VPBox [202], an Android OS-level sandbox framework via container-based virtualization that overcame the limitations of the previous works, integrating with the principle of anti-evasion. In particular, VPBox offers complete device virtualization for all the device components (e.g., WiFi, Camera), minimizing the artifacts in the VPs, and it can customize the virtual device configurations (e.g., OS version) for each VP.

6.6.1 *DroidDungeon* vs. SOTA

Emulators are programs that simulate the functionality of some hardware, thus providing great scalability and flexibility. For instance, the virtual environment can be restored to a clean snapshot in seconds. However, the anti-evasion criterion is demanded to the sandbox

itself, which has to implement the "bypass" logic for each evasive technique. In particular, DroidScope, CopperDroid, and DroidBox do not enforce any detection resilience mechanism, while hook-based techniques can bypass only a subset of evasive controls, leaving several artifacts uncovered. For instance, Xposed is not designed to hook into lower-level system calls; hence, an attacker can detect the emulator by making direct syscalls. *DroidDungeon* leverages the probe mechanism, which allows the hooking of user functions and system calls. Thus, we can provide more fine-grained analysis mechanisms that only hook the least possible set of functions. For instance, to identify the opening of a file, *DroidDungeon* hooks only file-related system calls, avoiding hooking all the high-level functions for both the Java and the native layers.

Moreover, all the container-based virtualization techniques and framework-level modifications (e.g., RealDroid) heavily modify the Android kernel and OS layers, which can make integration with system updates challenging (no maintainability criterion); in addition, the former techniques share the assumption to be executed on an actual phone affecting scalability.

Finally, it is essential to note that the primary goal of container-based virtualization is to create an isolated environment. Still, more is needed to provide a suitable technique for monitoring and analyzing the app's behavior. On the contrary, *DroidDungeon* can be deployed in both an emulated and actual device. In the former case, *DroidDungeon* has to enforce the anti-evasion criterion. At the same time, the second one leverages the underlying actual device to bypass the evasive controls, behaving like a virtual phone. Moreover, we developed a fully separate kernel module, which can be easily integrated with newer Android versions.

Table 6.6 compares *DroidDungeon* and the other state-of-the-art solutions based on the anti-evasion, scalability, and maintainability criteria. At the time of writing, our solution remains the best tool to analyze Android malware dynamically.

Table 6.6 Comparison between *DroidDungeon* and SOTA w.r.t. the anti-evasion, maintainability, and scalability criteria.

Requirement	**Container-Based**			**Emulator-Based**			
	Cells [14]	Condroid [240]	VPBox [202]	VMI-based [243, 211]	Framework mod. [170, 134]	Hook-based [137, 62, 49, 32, 74]	*DroidDungeon*
Anti-evasion	◐	◐	●	○	●	◐	●
Maintenability	○	○	○	◐	○	◐	●
Scalability	○	○	○	◐	●	●	●

6.7 Discussion

The first limitation of this work is related to the fact that the probe mechanisms cannot measure direct memory access or reading Java object fields. For instance, simple evasive checks aim to verify the fields of the `Build` class. We cannot measure these events even if *DroidDungeon* can mitigate them by updating the `Build` class field values when a new app starts.

Second, when a probe is placed in a userspace program, the instruction at the probed location is overwritten by a jump to the handler routine. This mechanism introduces artifacts into the memory of the probed functions that an attacker could exploit to detect the sandbox.

Third, Garfinkel et al. [76] demonstrated how making hardware emulation and native hardware indistinguishable is fundamentally infeasible. The emulator-based implementation of *DroidDungeon* inherits all limitations of the hardware emulation, such as time delays during the execution of certain instructions. However, we recall that *DroidDungeon* can also be distributed on an actual device.

Fourth, we collected the events by stimulating each app with ARES. Thus, we inherit the limitations of the dynamic analysis [5, 169], namely, there are parts of the code that may not be explored.

Finally, as mentioned in Section 6.5.3, our measures in comparing execution traces should be considered a lower bound because there are evasive controls that our sandbox does not need to mitigate. In those cases, if there are differences, they are not measured.

Chapter 7

Future Work and Conclusion

7.1 Future work

This thesis discusses several aspects of surreptitious code and the protection mechanisms that Android apps – both malicious and benign – employ. On one side, benign actors have to protect their apps from several attack vectors, such as man-at-the-end attacks, by routinely developing more robust methodologies. Conversely, malware authors aim to fly under the radar of detection and analysis tools by exploiting anti-static and anti-behavioral analysis techniques. Even if my thesis aims to contribute to the cat-and-mouse game between goodware and malware, the challenges that affect protection mechanisms from benign and malicious standpoints still need to be solved. In addition, this research topic is constantly evolving: malware and goodware authors regularly improve their protection schemes to make them more and more effective in reacting to novel attack vectors or analysis systems. However, the research presented in this thesis suggests several avenues for future directions. In particular, this section highlights two possible tracks to build upon our work.

The first track consists of improving the security posture and awareness of Android apps and their developers. Nowadays, developers do not only have to worry about developing secure code but also about protecting their apps from reverse engineering and code tampering. In particular, according to the OWASP Mobile Top Ten [165], the seventh issue for Android apps is insufficient binary protection. It is worth noticing that *all* apps are vulnerable to binary attacks (e.g., repackaging). Thus, in this context, the challenge lies in the fact that if there are additional protective measures (e.g., obfuscation for static analysis and APK tampering verification for code manipulation), successful attacks become more complex and more challenging to achieve, leading the attacker to give up if it is too complicated.

From the benign standpoint, the problem of security awareness is twofold. On one side, several developers are unaware of and do not include any protection mechanisms in their apps against these issues. Conversely, some remaining benign apps include ineffective protection techniques or consider only part of the problem. For instance, as highlighted in Section 6.5, goodware tends to rely on "homemade" evasive techniques instead of taking advantage of reliable services like the Play Integrity API offered by Google.

In tandem with the security awareness, a technological aspect should be considered. The Android operating system constantly evolves, and new features are routinely introduced, which can impact the robustness or effectiveness of current protection techniques. For instance, this thesis has investigated the impact of the Android app-level virtualization in the repackaging attack and proposed a novel protection scheme. Moreover, novel features can also affect the operating system's security. For instance, in Chapter 5, we discussed a novel state inference attack that has remained dormant since the inotify APIs were added to the system. Thus, future works should consider how novel features affect the security posture of modern Android apps and the entire operating system.

The second track consists of continuously improving the malware understanding and the effectiveness of the security tools for both static and dynamic analysis. For instance, Chapter 4 highlights how the state-of-the-art static analysis tools almost ignore the behavior exposed by the compiled native libraries. This work has contributed to this cat-and-mouse game, and we hope that our suspicious tag will be the building block for future research on Android malware and its detection. We suppose that exciting applications can be found in machine-learning-based analysis systems, and including native features could improve their robustness against these kinds of cheats.

Finally, an obvious extension of our work is the continuous development of *DroidDungeon* (introduced in Section 6.3) to improve its effectiveness concerning the anti-evasion criterion in large-scale malware analysis. Moreover, this dynamic analysis tool should be enhanced to understand other important aspects of anti-analysis techniques that malicious actors exploit. For instance, inspecting anti-static analysis techniques is an orthogonal research topic concerning the anti-behavioral analysis we conducted during this thesis. It is well known, and our results highlighted that modern malware (ab)uses DCL to load potentially harmful code at runtime. However, an in-depth analysis of the dynamically loaded code is still needed; for instance, exploring the type of loaded files, when malicious actors exploit these mechanisms, and the behavior they expose would be interesting.

7.2 Conclusion

The various projects that we tackled and discussed in this thesis allowed us to analyze numerous issues related to the security of the Android ecosystem. This research allowed us to understand, by changing the point of view for the analysis, how protection and anti-analysis techniques are not always considered an integral piece of the security of the Android ecosystem. Ensuring the security of the entire ecosystem becomes a very challenging task: different actors at play, with additional requirements and security constraints, make a generic approach to security ineffective.

In Chapter 3, this thesis starts with a security analysis of modern repackaging attacks, aiming at proposing a novel protection scheme – *MARVEL*– able to protect Android apps against traditional and virtualization-based repackaging attacks. We enforced the *MARVEL* protection scheme by I) *MARVELoid*, which injects the protections inside each plugin app, and II) the trusted container, a customized version of the VirtualApp framework to create a trusted execution environment and enforce the protection at runtime. We designed *MARVELoid* to minimize the impact on the end-user by only requiring the device to be equipped with the TC without any modification to the underlying OS. From an experimental point of view, our evaluation of *MARVEL* over $4,000$ Android apps demonstrated the applicability and efficacy of the tool and the proposed protection scheme.

Then, Chapter 4 investigates different aspects related to the usage of native code in Android apps by performing the first longitudinal analysis of the use of native components. This allowed us to identify several suspicious uses related to the JNI code. Moreover, it showed how our automatically assigned suspicious tags could pinpoint the code region to inspect and speed up the analysis process of Android malware, studying the behavior for all supported architectures.

The analysis of native code has brought to light a novel state inference attack based on the inotify APIs discussed in Chapter 5. Then, thanks to inotify, it is possible to execute code promptly to mount more complex attacks, such as phishing. Moreover, Chapter 5 introduced the concepts of file system footprint and signature and showed how these "traces" lead to figuring out when a particular app is starting. Interestingly, this work also opens up exciting future works because these concepts can be ported to different operating systems.

Our measurements show that *all* Android apps are vulnerable if the attacker can monitor a file in the installation path. Otherwise, this chapter has shown that it is still possible to carry it out on a smaller but significant number of apps. What is more, the attacker can unleash it from a malicious app without requesting any permission. Fortunately, we have also shown

the existence of practical remediation. However, even assuming they are all implemented immediately, the number of devices left vulnerable will remain high for many years because of the timeframe to deploy these updates, and many device vendors will not apply them.

Finally, Chapter 6 focuses on seeking, collecting, and measuring Android evasive techniques based on their behaviors and methods, both in malicious and benign apps. For this purpose, it introduces *DroidDungeon*, a probe-based sandbox that jointly fulfills the anti-evasion, maintainability, and scalability criteria. The experiments show the primary purposes of evasive checks in malware and goodware, and our main result highlights that 14% of malware and 4% of goodware refrain from running if their evasive controls detect a potential analysis environment. It is crucial to consider these percentages when dealing with dynamic analysis of Android apps and, thus, consider the bias introduced by anti-analysis and evasive techniques, which this thesis sought to shed light on.

Bibliography

[1] Jiagu. http://jiagu.360.cn/, 2023. Accessed online: January 26, 2024.

[2] Y. Aafer, G. Tao, J. Huang, X. Zhang, and N. Li. Precise android api protection mapping derivation and reasoning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1151–1164, 2018.

[3] V. Afonso, A. Bianchi, Y. Fratantonio, A. Doupé, M. Polino, P. de Geus, C. Kruegel, and G. Vigna. Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In *The Network and Distributed System Security Symposium*, pages 1–15, 2016.

[4] V. Afonso, A. Kalysch, T. Müller, D. Oliveira, A. Grégio, and P. L. de Geus. Lumus: Dynamically uncovering evasive android applications. In *Information Security: 21st International Conference, ISC 2018, Guildford, UK, September 9–12, 2018, Proceedings 21*, pages 47–66. Springer, 2018.

[5] A. Aggarwal and P. Jalote. Integrating static and dynamic analysis for detecting vulnerabilities. In *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, volume 1, pages 343–350. IEEE, 2006.

[6] W. H. Ahn, S. Park, J. Oh, and S.-H. Lim. Inishing: a ui phishing attack to exploit the vulnerability of inotify in android smartphones. *IEICE TRANSACTIONS on Information and Systems*, 99(9):2404–2409, 2016.

[7] A. V. Aho, R. Sethi, and J. D. Ullman. Compilers, principles, techniques. *Addison wesley*, 7(8):9, 1986.

[8] S. Alam, Z. Qu, R. Riley, Y. Chen, and V. Rastogi. Droidnative: Automating and optimizing detection of android native code malware variants. *computers & security*, 65:230–246, 2017.

[9] M. Alecci, R. Cestaro, M. Conti, K. Kanishka, and E. Losiouk. Mascara: A novel attack leveraging android virtualization. *arXiv preprint arXiv:2010.10639*, 2020.

[10] V. Alessio, R. Leonardo, B. Cataldo, T. Marco, C. Mariano, and T. Paolo. Empirical assessment of the effort needed to attack programs protected with client/server code splitting. *Empirical Software Engineering*, 25(1):1–48, 2020.

[11] A. Ali-Gombe, S. Sudhakaran, A. Case, G. G. Richard III, S. Zhu, P. Han, T. Kesavadas, D. Gu, K. Zhang, X. Wang, et al. Droidscraper: a tool for android in-memory object recovery and reconstruction. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*, pages 547–559, 2019.

[12] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. Androzoo: Collecting millions of android apps for the research community. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 468–471. IEEE, 2016.

[13] S. B. Andarzian and B. T. Ladani. Compositional taint analysis of native codes for security vetting of android applications. In *2020 10th International Conference on Computer and Knowledge Engineering (ICCKE)*, pages 567–572. IEEE, 2020.

[14] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh. Cells: a virtual mobile smartphone architecture. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 173–187, 2011.

[15] <Anonymized>. Android al-khaser. https://anonymous.4open.science/r/AAl-Khaser-10A0, 2023. Accessed online: January 26, 2024.

[16] S. Aonzo, A. Merlo, G. Tavella, and Y. Fratantonio. Phishing attacks on modern android. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1788–1801, 2018.

[17] S. Aonzo, G. C. Georgiu, L. Verderame, and A. Merlo. Obfuscapk: An open-source black-box obfuscation tool for android apps. *SoftwareX*, 11:100403, 2020.

[18] S. Aonzo, Y. Han, A. Mantovani, and D. Balzarotti. Humans vs. machines in malware classification. *Proc. of USENIX-23*, 2023.

[19] AppLovin. Applovin max. https://www.applovin.com/, 2023. Accessed online: January 26, 2024.

[20] AppMetrica. Appmetrica yadex. https://appmetrica.yandex, 2022. Accessed online: January 26, 2024.

[21] archer29m. Soot issue 1151. https://github.com/soot-oss/soot/issues/1151, 2019. Accessed online: January 26, 2024.

[22] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[23] Avast. Malware posing as dual instance app steals users' twitter credentials. https://blog.avast.com/malware-posing-as-dual-instance-app-steals-users-twitter-credentials, 2016. Accessed online: January 26, 2024.

[24] M. Backes, S. Bugiel, O. Schranz, P. von Styp-Rekowsky, and S. Weisgerber. Artist: The android runtime instrumentation and security toolkit. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 481–495. IEEE, 2017.

[25] BBC. Fake whatsapp app downloaded more than one million times. https://www.bbc.com/news/technology-41886157, 2017. Accessed online: January 26, 2024.

[26] L. Bello and M. Pistoia. Ares: triggering payload of evasive android malware. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, pages 2–12, 2018.

[27] H. Berger, C. Hajaj, and A. Dvir. Evasion is not enough: A case study of android malware. In *International Symposium on Cyber Security Cryptography and Machine Learning*, pages 167–174. Springer, 2020.

[28] N. Bergman. Android anti-hooking techniques in java. https://d3adend.org/blog/posts/android-anti-hooking-techniques-in-java/, 2015. Accessed online: January 26, 2024.

[29] S. Berlato and M. Ceccato. A large-scale study on the adoption of anti-debugging and anti-tampering protections in android apps. *Journal of Information Security and Applications*, 52:102463, 2020.

[30] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna. What the app is that? deception and countermeasures in the android user interface. In *2015 IEEE Symposium on Security and Privacy*, pages 931–948. IEEE, 2015.

[31] BleepingComputer. Fortnite android app vulnerable to man-in-the-disk attacks. https://www.bleepingcomputer.com/news/security/fortnite-android-app-vulnerable-to-man-in-the-disk-attacks/, 2018. Accessed online: January 26, 2024.

[32] L. Bordoni, M. Conti, and R. Spolaor. Mirage: Toward a stealthier and modular malware analysis sandbox for android. In *Computer Security–ESORICS 2017: 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part I 22*, pages 278–296. Springer, 2017.

[33] brevent. genuine. https://github.com/brevent/genuine, 2023. Accessed online: January 26, 2024.

[34] c3r34lk1ll3r. Cve-2019-2215 exploit. https://github.com/c3r34lk1ll3r/CVE-2019-2215, 2020. Accessed online: January 26, 2024.

[35] D. Caputo, L. Verderame, S. Aonzo, and A. Merlo. Droids in disarray: detecting frame confusion in hybrid android apps. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 121–139. Springer, 2019.

[36] M. Ceccato, M. Dalla Preda, J. Nagra, C. Collberg, and P. Tonella. Barrier slicing for remote software trusting. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 27–36. IEEE, 2007.

[37] Chartboost. Chartboost. https://support.chartboost.com/en, 2023. Accessed online: January 26, 2024.

[38] K. Chen, Y. Zhang, and P. Liu. Leveraging information asymmetry to transform android apps into self-defending code against repackaging attacks. *IEEE Transactions on Mobile Computing*, 17(8):1879–1893, 2017.

[39] Q. A. Chen, Z. Qian, and Z. M. Mao. Peeking into your app without actually seeing it:{UI} state inference and novel android attacks. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 1037–1052, 2014.

[40] S. F. Conservancy. Qemu. https://www.qemu.org/, 2023. Accessed online: January 26, 2024.

[41] T. M. Corporation. Cve-2011-1823. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-1823, 2011. Accessed online: January 26, 2024.

[42] T. M. Corporation. Cve-2014-3153. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3153, 2014. Accessed online: January 26, 2024.

[43] T. M. Corporation. Cve-2016-5195. https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2016-5195, 2016. Accessed online: January 26, 2024.

[44] T. M. Corporation. Cve-2019-2215. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-2215, 2019. Accessed online: January 26, 2024.

[45] T. M. Corporation. Software discovery. https://attack.mitre.org/techniques/T1418/, 2022. Accessed online: January 26, 2024.

[46] T. M. Corporation. mbc-markdown. https://github.com/MBCProject/mbc-markdown, 2023. Accessed online: January 26, 2024.

[47] T. M. Corporation. Evademe. https://attack.mitre.org/techniques/T1633/001/, 2023. Accessed online: January 26, 2024.

[48] Cryptomathic. Virtualization/sandbox evasion: System checks. https://www.cryptomathic.com/news-events/blog/app-hardening-for-mobile-banking-and-payment-apps-emulator-detection, 2022. Accessed online: January 26, 2024.

[49] Y. Cui, Y. Sun, and Z. Lin. Droidhook: a novel api-hook based android malware dynamic analysis sandbox. *Automated Software Engineering*, 30(1):10, 2023.

[50] D. Dai, R. Li, J. Tang, A. Davanian, and H. Yin. Parallel space traveling: A security analysis of app-level virtualization in android. In *Proceedings of the 25th ACM Symposium on Access Control Models and Technologies*, pages 25–32, 2020.

[51] S. Dambra, Y. Han, S. Aonzo, P. Kotzias, A. Vitale, J. Caballero, D. Balzarotti, and L. Bilge. Decoding the secrets of machine learning in malware classification: A deep dive into datasets, feature extraction, and model performance. *arXiv preprint arXiv:2307.14657*, 2023.

[52] darvincisec. Anti debug and memory dump. https://github.com/darvincisec/AntiDebugandMemoryDump, 2021. Accessed online: January 26, 2024.

[53] G. Developers. Enable multidex for apps with over 64k methods. https://developer.android.com/studio/build/multidex, 2020. Accessed online: January 26, 2024.

[54] DimitriFourny. Cve-2019-2215 exploit. https://github.com/DimitriFourny/cve-2019-2215, 2020. Accessed online: January 26, 2024.

[55] D. C. D'Elia, E. Coppa, F. Palmaro, and L. Cavallaro. On the dissection of evasive malware. *IEEE Transactions on Information Forensics and Security*, 15:2750–2765, 2020.

[56] EFForg. apkeep. https://github.com/EFForg/apkeep, 2023. Accessed online: January 26, 2024.

[57] W. F. Elsersy, A. Feizollah, and N. B. Anuar. The rise of obfuscated android malware and impacts on detection methods. *PeerJ Computer Science*, 8:e907, 2022.

[58] erev0s. 3 ways to detect the selinux status in android natively. https://erev0s.com/blog/3-ways-detect-selinux-status-android-natively/, 2020. Accessed online: January 26, 2024.

[59] ESET. Badbazaar espionage tool targets android users via trojanized signal and telegram apps. https://www.welivesecurity.com/en/eset-research/badbazaar-espionage-tool-targets-android-users-trojanized-signal-telegram-apps/, 2023. Accessed online: January 26, 2024.

[60] evilthreads669966. Evademe. https://github.com/evilthreads669966/evademe, 2021. Accessed online: January 26, 2024.

[61] facundoolano. Google play scraper. https://github.com/facundoolano/google-play-scraper, 2022. Accessed online: January 26, 2024.

[62] F. Faghihi, M. Zulkernine, and S. Ding. Camodroid: An android application analysis environment resilient against sandbox evasion. *Journal of Systems Architecture*, 125: 102452, 2022.

[63] P. Faruki, R. Bhan, V. Jain, S. Bhatia, N. El Madhoun, and R. Pamula. A survey and evaluation of android-based malware evasion techniques and detection frameworks. *Information*, 14(7):374, 2023.

[64] A. P. Felt and D. Wagner. Phishing on mobile devices. 2011.

[65] E. Fernandes, Q. A. Chen, J. Paupore, G. Essl, J. A. Halderman, Z. M. Mao, and A. Prakash. Android ui deception revisited: Attacks and defenses. In *International*

Conference on Financial Cryptography and Data Security, pages 41–59. Springer, 2016.

[66] F. File. Firmware file. firmwarefile.com. Accessed online: January 26, 2024.

[67] Flurry. Flurry. https://www.flurry.com/, 2023. Accessed online: January 26, 2024.

[68] Flutter. Flutter. https://flutter.dev/, 2023. Accessed online: January 26, 2024.

[69] S. C. Foundation. Cuckoo sandbox. https://cuckoosandbox.org/, 2023. Accessed online: January 26, 2024.

[70] G. Fourtounis, L. Triantafyllou, and Y. Smaragdakis. Identifying java calls in native code via binary scanning. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 388–400, 2020.

[71] Y. Fratantonio, C. Qian, S. P. Chung, and W. Lee. Cloak and dagger: from two permissions to complete control of the ui feedback loop. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 1041–1057. IEEE, 2017.

[72] L. Futcher and R. Von Solms. Guidelines for secure software development. In *Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries: riding the wave of technology*, pages 56–65, 2008.

[73] Fuzion24. Android hostile environment detection. https://github.com/Fuzion24/AndroidHostileEnvironmentDetection, 2016. Accessed online: January 26, 2024.

[74] J. Gajrani, J. Sarswat, M. Tripathi, V. Laxmi, M. S. Gaur, and M. Conti. A robust dynamic analysis system preventing sandbox detection by android malware. In *Proceedings of the 8th International Conference on Security of Information and Networks*, pages 290–295, 2015.

[75] N. Galloro, M. Polino, M. Carminati, A. Continella, and S. Zanero. A systematical and longitudinal study of evasive behaviors in windows malware. *Computers & Security*, 113:102550, 2022.

[76] T. Garfinkel, K. Adams, A. Warfield, J. Franklin, et al. Compatibility is not transparency: Vmm detection myths and realities. In *HotOS*, 2007.

[77] Genymobile. Genymotion. https://www.genymotion.com/, 2023. Accessed online: January 26, 2024.

[78] G. C. Georgiu. Playstore dowloader. https://github.com/ClaudiuGeorgiu/PlaystoreDownloader, 2022. Accessed online: January 26, 2024.

[79] Ghidra. Ghidra. https://ghidra.re/, 2023. Accessed online: January 26, 2024.

[80] H. Gonzalez, N. Stakhanova, and A. A. Ghorbani. Droidkin: Lightweight detection of android apps similarity. In *International Conference on Security and Privacy in Communication Networks: 10th International ICST Conference, SecureComm 2014, Beijing, China, September 24-26, 2014, Revised Selected Papers, Part I 10*, pages 436–453. Springer, 2015.

[81] Google. Bitunmap: Attacking android ashmem. https://googleprojectzero.blogspot.com/2016/12/bitunmap-attacking-android-ashmem.html, 2017. Accessed online: January 26, 2024.

[82] Google. Protecting webview with safe browsing. https://android-developers.googleblog.com/2018/04/protecting-webview-with-safe-browsing.html, 2018. Accessed online: January 26, 2024.

[83] Google. Android use-after-free in binder. https://googleprojectzero.github.io/0days-in-the-wild/0day-RCAs/2019/CVE-2019-2215.html, 2020. Accessed online: January 26, 2024.

[84] Google. Android 12.1 – selinux hooks. https://android.googlesource.com/kernel/common/+/refs/tags/android-12.1.0_r0.35/security/selinux/hooks.c, 2021. Accessed online: January 26, 2024.

[85] Google. The activity lifecycle. https://developer.android.com/guide/components/activities/activity-lifecycle, 2022. Accessed online: January 26, 2024.

[86] Google. Android app bundle. https://developer.android.com/guide/app-bundle, 2022. Accessed online: January 26, 2024.

[87] Google. Declaring package visibility needs. https://developer.android.com/training/package-visibility/declaring, 2022. Accessed online: January 26, 2024.

[88] Google. Android scoped storage. https://developer.android.com/training/data-storage#scoped-storage, 2022. Accessed online: January 26, 2024.

[89] Google. Security updates and resources – severity. https://source.android.com/security/overview/updates-resources#severity, 2022. Accessed online: January 26, 2024.

[90] Google. Android mainline – selinux macro. https://android.googlesource.com/platform/system/sepolicy/+/refs/heads/master/public/te_macros, 2022. Accessed online: January 26, 2024.

[91] Google. Create a custom notification layout. https://developer.android.com/training/notify-user/custom-notification, 2022. Accessed online: January 26, 2024.

[92] Google. Fileobserver. https://developer.android.com/reference/android/os/FileObserver, 2022. Accessed online: January 26, 2024.

[93] Google. Average number of apps installed on users' smartphones. https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/average-number-of-apps-on-smartphones/, 2022. Accessed online: January 26, 2024.

[94] Google. Android 8.0 behavior changes. https://developer.android.com/about/versions/oreo/android-8.0-changes, 2023. Accessed online: January 26, 2024.

[95] Google. Android abis. https://developer.android.com/ndk/guides/abis, 2023. Accessed online: January 26, 2024.

[96] Google. Android app categories. https://support.google.com/googleplay/android-developer/answer/9859673, 2023. Accessed online: January 26, 2024.

[97] Google. Android package visibility. https://developer.android.com/training/package-visibility, 2023. Accessed online: January 26, 2024.

[98] Google. Permissions on android. https://developer.android.com/guide/topics/permissions/overview, 2023. Accessed online: January 26, 2024.

[99] Google. Configuring art. https://source.android.com/docs/core/runtime/configure, 2023. Accessed online: January 26, 2024.

[100] Google. Developer guides. https://developer.android.com/guide, 2023. Accessed online: January 26, 2024.

[101] Google. Overview of google play services. https://developers.google.com/android/guides/overview, 2023. Accessed online: January 26, 2024.

[102] Google. Play integrity api. https://developer.android.com/google/play/integrity, 2023. Accessed online: January 26, 2024.

[103] Google. Android linker source code, call_constructors method. https://android.googlesource.com/platform/bionic/+/master/linker/linker_soinfo.cpp#516, 2023. Accessed online: January 26, 2024.

[104] Google. Protect against security threats with safetynet. https://developer.android.com/training/safetynet, 2023. Accessed online: January 26, 2024.

[105] Google. Android verified boot 2.0. https://android.googlesource.com/platform/external/avb/+/master/README.md, 2023. Accessed online: January 26, 2024.

[106] Google. Google safe browsing service. https://developer.android.com/develop/ui/views/layout/webapps/managing-webview#safe-browsing, 2023. Accessed online: January 26, 2024.

[107] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard. Information flow analysis of android applications in droidsafe. In *NDSS*, volume 15, page 110, 2015.

[108] S. R. Group. Soot - a java optimization framework. https://github.com/soot-oss/soot, 2023. Accessed online: January 26, 2024.

[109] GToad. Android anti debug. https://github.com/GToad/Android_Anti_Debug, 2018. Accessed online: January 26, 2024.

[110] Guardsquare. Mobile application protection. https://www.guardsquare.com/, 2023. Accessed online: January 26, 2024.

[111] S. Hazarika. Xposed. https://www.xda-developers.com/best-xposed-modules/, 2022. Accessed online: January 26, 2024.

[112] 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.

[113] M. Ibrahim, A. Imran, and A. Bianchi. Safetynot: on the usage of the safetynet attestation api in android. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, pages 150–162, 2021.

[114] InMobi. Inmobi. https://www.inmobi.com/sdk, 2023. Accessed online: January 26, 2024.

[115] I. Innovations. Parallel accounts. https://play.google.com/store/apps/details?id=com.in.parallel.accounts, 2023. Accessed online: January 26, 2024.

[116] Irdeto. Denuvo mobile games protection. https://irdeto.com/denuvo/mobile-games-protection/, 2023. Accessed online: January 26, 2024.

[117] j0nk0. Android dirtycow. https://github.com/j0nk0/GetRoot-Android-DirtyCow, 2019. Accessed online: January 26, 2024.

[118] S. Jamalpur, Y. S. Navya, P. Raja, G. Tagore, and G. R. K. Rao. Dynamic malware analysis using cuckoo sandbox. In *2018 Second international conference on inventive communication and computational technologies (ICICCT)*, pages 1056–1060. IEEE, 2018.

[119] Y. Jing, Z. Zhao, G.-J. Ahn, and H. Hu. Morpheus: automatically generating heuristics to detect android emulators. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 216–225, 2014.

[120] kangtastic. Cve-2019-2215 exploit. https://github.com/kangtastic/cve-2019-2215, 2019. Accessed online: January 26, 2024.

[121] Kaspersky. Malware in august: One year after the first android malware emerged, & the clones of zeus. https://www.kaspersky.com/about/press-releases/2011_malware-in-august-one-year-after-the-first-android-malware-emerged--the-clones-of-zeus, 2011. Accessed online: January 26, 2024.

[122] T. kernel development community. Bpf documentation — the linux kernel documentation. https://www.kernel.org/doc/html/latest/bpf/index.html, 2022. Accessed online: January 26, 2024.

[123] M. Kerrisk. proc.5. https://man7.org/linux/man-pages/man5/proc.5.html, 2021. Accessed online: January 26, 2024.

[124] M. Kerrisk. xdr. https://man7.org/linux/man-pages/man3/xdr.3.html, 2021. Accessed online: January 26, 2024.

[125] M. Kerrisk. readelf. https://man7.org/linux/man-pages/man1/readelf.1.html, 2023. Accessed online: January 26, 2024.

[126] S. Kevin. Bankbot found on google play and targets ten new uae banking apps. https://www.trendmicro.com/en_us/research/17/i/bankbot-found-google-play-targets-ten-new-uae-banking-apps.html, 2017. Accessed online: January 26, 2024.

[127] Killuaa27. Soot issue 1474. https://github.com/soot-oss/soot/issues/1474, 2020. Accessed online: January 26, 2024.

[128] B. Kondracki, B. A. Azad, N. Miramirkhani, and N. Nikiforakis. The droid is in the details: Environment-aware evasion of android sandboxes. In *Proceedings of the 29th Network and Distributed System Security Symposium (NDSS)*, 2022.

[129] A. K. Lab. Mobile malware evolution. https://securelist.com/mobile-malware-evolution-2021/105876/, 2022. Accessed online: January 26, 2024.

[130] S. Lee. Jni program analysis with automatically extracted c semantic summary. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 448–451, 2019.

[131] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 280–291. IEEE, 2015.

[132] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and L. Traon. Static analysis of android apps: A systematic literature review. *Information and Software Technology*, 88:67–95, 2017.

[133] L. Li, T. F. Bissyandé, and J. Klein. Rebooting research on detecting repackaged android apps: Literature review and benchmark. *IEEE Transactions on Software Engineering*, 47(4):676–693, 2019.

[134] L. Liu, Y. Gu, Q. Li, and P. Su. Realdroid: Large-scale evasive malware detection on" real devices". In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–8. IEEE, 2017.

[135] LordNoteworthy. Al-khaser. https://github.com/LordNoteworthy/al-khaser, 2023. Accessed online: January 26, 2024.

[136] R. Love. inotify documentation. https://www.kernel.org/doc/Documentation/filesystems/inotify.txt, 2015. Accessed online: January 26, 2024.

[137] C. P. S. T. LTD. Cuckoodroid. https://github.com/idanr1986/cuckoo-droid, 2017. Accessed online: January 26, 2024.

[138] C. P. S. T. Ltd. Man-in-the-disk: A new attack surface for android apps. https://blog.checkpoint.com/2018/08/12/man-in-the-disk-a-new-attack-surface-for-android-apps/, 2018. Accessed online: January 26, 2024.

[139] J. L. N. T. C. Ltd. Virtualapp, 2020. URL https://github.com/asLody/VirtualApp. Accessed online: January 26, 2024.

[140] L. Luo, Y. Fu, D. Wu, S. Zhu, and P. Liu. Repackage-proofing android apps. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 550–561. IEEE, 2016.

[141] T. Luo, C. Zheng, Z. Xu, and X. Ouyang. Anti-plugin: Don't let your app play as an android plugin. *Proceedings of Blackhat Asia*, 2017.

[142] Z. Ma, H. Ge, Y. Liu, M. Zhao, and J. Ma. A combination method for android malware detection based on control flow graphs and machine learning algorithms. *IEEE access*, 7:21235–21245, 2019.

[143] L. Maffia, D. Nisi, P. Kotzias, G. Lagorio, S. Aonzo, and D. Balzarotti. Longitudinal study of the prevalence of malware evasive techniques. *arXiv preprint arXiv:2112.11289*, 2021.

[144] D. Maier, T. Müller, and M. Protsenko. Divide-and-conquer: Why android malware cannot be stopped. In *2014 Ninth International Conference on Availability, Reliability and Security*, pages 30–39. IEEE, 2014.

[145] L. Malisa, K. Kostiainen, and S. Capkun. Detecting mobile application spoofing attacks by leveraging user visual similarity perception. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 289–300, 2017.

[146] Malwarebytes. Trojan.dropper. https://blog.malwarebytes.com/detections/trojan-dropper/, 2022. Accessed online: January 26, 2024.

[147] G. Meng, Y. Xue, C. Mahinthan, A. Narayanan, Y. Liu, J. Zhang, and T. Chen. Mystique: Evolving android malware for auditing anti-malware tools. In *Proceedings*

of the 11th ACM on Asia conference on computer and communications security, pages 365–376, 2016.

[148] A. Merlo, A. Ruggia, L. Sciolla, and L. Verderame. Armand: Anti-repackaging through multi-pattern anti-tampering based on native detection. *Pervasive and Mobile Computing*, 76:101443, 2021.

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

[150] N. Miramirkhani, M. P. Appini, N. Nikiforakis, and M. Polychronakis. Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 1009–1024.

[151] S. Mirza, H. Abbas, W. B. Shahid, N. Shafqat, M. Fugini, Z. Iqbal, and Z. Muhammad. A malware evasion technique for auditing android anti-malware solutions. In *2021 IEEE 30th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 125–130. IEEE, 2021.

[152] muellerberndt. frida-detection. https://github.com/muellerberndt/frida-detection, 2022. Accessed online: January 26, 2024.

[153] J. Nagra and C. Collberg. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Pearson Education, 2009.

[154] F. Naves, A. Conway, S. W. Jones, and A. Mcneil. Tanglebot: New advanced sms malware targets mobile users across u.s. and canada with covid-19 lures. https://www.cloudmark.com/en/blog/malware/tanglebot-new-advanced-sms-malware-targets-mobile-users-across-us-and-canada-covid-19. Accessed online: January 26, 2024.

[155] ndeztea. Appium issue 12555. https://github.com/appium/appium/issues/12555, 2019. Accessed online: January 26, 2024.

[156] D. Nisi, A. Bianchi, and Y. Fratantonio. Exploring {Syscall-Based} semantics reconstruction of android applications. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 517–531, 2019.

[157] NowSecure. Frida. https://frida.re/, 2023. Accessed online: January 26, 2024.

[158] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pages 543–558, 2013.

[159] Oracle. Jarsigner. https://docs.oracle.com/javase/7/docs/technotes/tools/windows/jarsigner.html, 2021. Accessed online: January 26, 2024.

[160] Oracle. Jni types and data structures. https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/types.html, 2023. Accessed online: January 26, 2024.

[161] Oracle. Dynamic proxy classes. https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html, 2023. Accessed online: January 26, 2024.

[162] Oracle. Jni functions. https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html, 2023. Accessed online: January 26, 2024.

[163] Oracle. Oracle jni. https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html, 2023. Accessed online: January 26, 2024.

[164] OWASP. M7: Insufficient binary protection. https://owasp.org/www-project-mobile-top-10/2023-risks/m7-insufficient-binary-protection.html, 2023. Accessed online: January 26, 2024.

[165] OWASP. Owasp mobile top 10. https://owasp.org/www-project-mobile-top-10/2023-risks/m7-insufficient-binary-protection.html, 2023. Accessed online: January 26, 2024.

[166] PAGalaxyLab. Yahfa. https://github.com/PAGalaxyLab/YAHFA, 2023. Accessed online: January 26, 2024.

[167] R. Paleari, L. Martignoni, G. F. Roglia, and D. Bruschi. A fistful of red-pills: How to automatically generate procedures to detect cpu emulators. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, volume 41, page 86, 2009.

[168] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis. Rage against the virtual machine: hindering dynamic analysis of android malware. In *Proceedings of the seventh european workshop on system security*, pages 1–6, 2014.

[169] A. Petukhov and D. Kozlov. Detecting security vulnerabilities in web applications using dynamic analysis with penetration testing. *Computing Systems Lab, Department of Computer Science, Moscow State University*, pages 1–120, 2008.

[170] pjlantz. Droidbox. https://github.com/pjlantz/droidbox, 2019. Accessed online: January 26, 2024.

[171] A. Possemato, S. Aonzo, D. Balzarotti, and Y. Fratantonio. Trust, But Verify: A Longitudinal Analysis Of Android OEM Compliance and Customization. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, May 2021.

[172] A. Possemato, D. Nisi, and Y. Fratantonio. Preventing and detecting state inference attacks on android. In *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS), Virtual, 21st-25th February*, 2021.

[173] M. Protsenko, S. Kreuter, and T. Müller. Dynamic self-protection and tamperproofing for android apps using native code. In *2015 10th International Conference on Availability, Reliability and Security*, pages 129–138. IEEE, 2015.

[174] C. Qian, X. Luo, Y. Shao, and A. T. Chan. On tracking information flows through jni in android applications. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 180–191. IEEE, 2014.

[175] Z. Qu, S. Alam, Y. Chen, X. Zhou, W. Hong, and R. Riley. Dydroid: Measuring dynamic code loading and its security implications in android applications. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 415–426. IEEE, 2017.

[176] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden. Harvesting runtime values in android applications that feature anti-analysis techniques. In *NDSS*, 2016.

[177] B. S. Rawal, R. K. Karne, and A. L. Wijesinha. Split protocol client/server architecture. In *2012 IEEE Symposium on Computers and Communications (ISCC)*, pages 000348–000353. IEEE, 2012.

[178] rednaga. Apkid. https://github.com/rednaga/APKiD, 2023. Accessed online: January 26, 2024.

[179] C. Ren, Y. Zhang, H. Xue, T. Wei, and P. Liu. Towards discovering and understanding task hijacking in android. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 945–959, 2015.

[180] C. Ren, P. Liu, and S. Zhu. Windowguard: Systematic protection of gui security in android. In *NDSS*, 2017.

[181] A. Romdhana, A. Merlo, M. Ceccato, and P. Tonella. Deep reinforcement learning for black-box testing of android apps. *ACM Transactions on Software Engineering and Methodology*, 2022.

[182] A. Ruggia. inotify-analyzer. https://gitlab.eurecom.fr/totoR13/inotify-analyzer, 2023. Accessed online: January 26, 2024.

[183] A. Ruggia, E. Losiouk, L. Verderame, M. Conti, and A. Merlo. Repack me if you can: An anti-repackaging solution based on android virtualization. In *Annual Computer Security Applications Conference*, pages 970–981, 2021.

[184] A. Ruggia, A. Possemato, S. Dambra, A. Merlo, S. Aonzo, and D. Balzarotti. The dark side of native code on android. 2022.

[185] A. Ruggia, A. Possemato, A. Merlo, D. Nisi, and S. Aonzo. Android, notify me when it is time to go phishing. In *EUROS&P 2023, 8th IEEE European Symposium on Security and Privacy*, 2023.

[186] A. Ruggia, D. Nisi, S. Dambra, A. Merlo, D. Balzarotti, and S. Aonzo. Unmasking the veiled: A comprehensive analysis of android evasive malware. 2024.

[187] O. Sahin, A. K. Coskun, and M. Egele. Proteus: Detecting android emulators from instruction-level profiles. In *Research in Attacks, Intrusions, and Defenses: 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings 21*, pages 3–24. Springer, 2018.

[188] J. Samhi, J. Gao, N. Daoudi, P. Graux, H. Hoyez, X. Sun, K. Allix, T. F. Bissyandé, and J. Klein. Jucify: A step towards android code unification for enhanced static analysis. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 1232–1244. IEEE, 2022.

[189] samohyes. Anti-vm-in-android. https://github.com/samohyes/Anti-vm-in-android, 2018. Accessed online: January 26, 2024.

[190] Samsung. Samsung knox. https://www.samsungknox.com/en, 2023. Accessed online: January 26, 2024.

[191] scottyab. Rootbeer. https://github.com/scottyab/rootbeer, 2021. Accessed online: January 26, 2024.

[192] S. Sebastián and J. Caballero. Avclass2: Massive malware tag extraction from av labels. In *Annual Computer Security Applications Conference*, pages 42–53, 2020.

[193] SecureList. Evil telegram doppelganger attacks chinese users. https://securelist.com/trojanized-telegram-mod-attacking-chinese-users/110482/, 2023. Accessed online: January 26, 2024.

[194] A. Security. libbpfgo. https://github.com/aquasecurity/libbpfgo, 2022. Accessed online: January 26, 2024.

[195] O. M. A. Security. Android anti-reversing defenses. https://mas.owasp.org/MASTG/Android/0x05j-Testing-Resiliency-Against-Reverse-Engineering/, 2023. Accessed online: January 26, 2024.

[196] O. M. A. Security. Android anti-reversing defenses. https://mas.owasp.org/MASTG/Android/0x05j-Testing-Resiliency-Against-Reverse-Engineering/, 2023. Accessed online: January 26, 2024.

[197] SentinelOne. Capratube – transparent tribe's caprarat mimics youtube to hijack android phones. https://www.sentinelone.com/labs/capratube-transparent-tribes-caprarat-mimics-youtube-to-hijack-android-phones/, 2023. Accessed online: January 26, 2024.

[198] L. Shi, J. Fu, Z. Guo, and J. Ming. " jekyll and hyde" is risky: Shared-everything threat mitigation in dual-instance apps. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, pages 222–235, 2019.

[199] L. Shi, J. Ming, J. Fu, G. Peng, D. Xu, K. Gao, and X. Pan. Vahunt: Warding off new repackaged android malware in app-virtualization's clothing. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 535–549, 2020.

[200] I. M. E. Software. evadroid. https://bitbucket.org/IBMmobile/evadroid/src/master/, 2018. Accessed online: January 26, 2024.

[201] L. Song, Z. Tang, Z. Li, X. Gong, X. Chen, D. Fang, and Z. Wang. Appis: Protect android apps against runtime repackaging attacks. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 25–32. IEEE, 2017.

[202] W. Song, J. Ming, L. Jiang, Y. Xiang, X. Pan, J. Fu, and G. Peng. Towards transparent and stealthy android os sandboxing via customizable container-based virtualization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2858–2874, 2021.

[203] R. Spreitzer, F. Kirchengast, D. Gruss, and S. Mangard. Procharvester: Fully automated analysis of procfs side-channel leaks on android. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 749–763, 2018.

[204] R. Spreitzer, G. Palfinger, and S. Mangard. Scandroid: Automated side-channel analysis of android apis. In *Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, pages 224–235, 2018.

[205] StatCounter. Mobile operating system market share worldwide. https://gs.statcounter.com/os-market-share/mobile/worldwide, 2023. Accessed online: January 26, 2024.

[206] StatCounter. Operating system market share worldwide. https://securelist.com/mobile-malware-evolution-2021/105876/, 2023. Accessed online: January 26, 2024.

[207] G. Stergiopoulos, D. Gritzalis, E. Vasilellis, and A. Anagnostopoulou. Dropping malware through sound injection: A comparative analysis on android operating systems. *Computers & Security*, 105:102228, 2021.

[208] stockrom.net. stockrom. stockrom.net. Accessed online: January 26, 2024.

[209] M. Sun, T. Wei, and J. C. Lui. Taintart: A practical multi-level information-flow tracking system for android runtime. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 331–342, 2016.

[210] S.-T. Sun, A. Cuadros, and K. Beznosov. Android rooting: Methods, detection, and evasion. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 3–14, 2015.

[211] K. Tam, A. Fattori, S. Khan, and L. Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *NDSS Symposium 2015*, pages 1–15, 2015.

[212] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro. The evolution of android malware and android analysis techniques. *ACM Computing Surveys (CSUR)*, 49(4):1–41, 2017.

[213] S. Tanner, I. Vogels, and R. Wattenhofer. Protecting android apps from repackaging using native code. In *Foundations and Practice of Security: 12th International Symposium, FPS 2019, Toulouse, France, November 5–7, 2019, Revised Selected Papers 12*, pages 189–204. Springer, 2020.

[214] D. Team. Droidplugin, 2020. URL https://github.com/DroidPluginTeam/DroidPlugin. Accessed online: January 26, 2024.

[215] H. Team. Hackingteam exploits. https://github.com/f47h3r/hackingteam_exploits/tree/master/android, 2015. Accessed online: January 26, 2024.

[216] M. Team. Multiple accounts:parallel app. https://play.google.com/store/apps/details?id=com.excelliance.multiaccounts, 2023. Accessed online: January 26, 2024.

[217] L. Tech. Parallel space - fmulti account. https://play.google.com/store/apps/details?id=com.lbe.parallel.intl, 2023. Accessed online: January 26, 2024.

[218] U. Technologies. Unity. https://unity.com/solutions/mobile/android-game-development, 2023. Accessed online: January 26, 2024.

[219] Tencent. Myapp – tencent. https://android.myapp.com/, 2022. Accessed online: January 26, 2024.

[220] R. Thomas. Android runtime restriction bypass. https://blog.quarkslab.com/android-runtime-restrictions-bypass.html, 2019. Accessed online: January 26, 2024.

[221] ThreatFabric. 300.000+ infections via droppers on google play store. https://threatfabric.com/blogs/deceive-the-heavens-to-cross-the-sea.html, 2021. Accessed online: January 26, 2024.

[222] K. Tian, D. Yao, B. G. Ryder, and G. Tan. Analysis of code heterogeneity for high-precision classification of repackaged malware. In *2016 IEEE Security and Privacy Workshops (SPW)*, pages 262–271. IEEE, 2016.

[223] topjohnwu. Magisk. https://github.com/topjohnwu/Magisk, 2023. Accessed online: January 26, 2024.

[224] totoR13. Soot issue 1615. https://github.com/soot-oss/soot/issues/1615, 2021. Accessed online: January 26, 2024.

[225] totoR13. Marvel. https://github.com/totoR13/MARVEL, 2023. Accessed online: January 26, 2024.

[226] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. 2010.

[227] T. Vidas and N. Christin. Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 447–458, 2014.

[228] VirusTotal. Virustotal. https://www.virustotal.com, 2023. Accessed online: January 26, 2024.

[229] Vungle. Vungle. https://support.vungle.com/hc/en-us/, 2022. Accessed online: January 26, 2024.

[230] F. Wang and Y. Shoshitaishvili. Angr-the next generation of binary analysis. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 8–9. IEEE, 2017.

[231] J. Wang, Y. Liu, C. Xu, X. Ma, and J. Lu. E-greendroid: effective energy inefficiency analysis for android applications. In *proceedings of the 8th Asia-Pacific Symposium on Internetware*, pages 71–80, 2016.

[232] D. Web. The coper — a new android banking trojan targeting colombian users. https://news.drweb.com/show/?i=14259&lng=en&c=5, 2021. Accessed online: January 26, 2024.

[233] D. Web. Android.spy.lydia trojans masquerade as an iranian online trading platform. https://news.drweb.com/show/?i=14748&lng=en, 2023. Accessed online: January 26, 2024.

[234] F. Wei, S. Roy, and X. Ou. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, pages 1329–1341, 2014.

[235] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou. Deep ground truth analysis of current android malware. In *International conference on detection of intrusions and malware, and vulnerability assessment*, pages 252–276. Springer, 2017.

[236] F. Wei, X. Lin, X. Ou, T. Chen, and X. Zhang. Jn-saf: Precise and efficient ndk/jni-aware inter-language static analysis framework for security vetting of android applications with native code. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1137–1150, 2018.

[237] D. Wu, D. Gao, R. H. Deng, and C. R. KC. When program analysis meets bytecode search: Targeted and efficient inter-procedural analysis of modern android apps in backdroid. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 543–554. IEEE, 2021.

[238] L. Wu, X. Du, and J. Wu. Mobifish: A lightweight anti-phishing scheme for mobile phones. In *2014 23rd International Conference on Computer Communication and Networks (ICCCN)*, pages 1–8. IEEE, 2014.

[239] Y. Wu, J. Huang, B. Liang, and W. Shi. Do not jail my app: Detecting the android plugin environments by time lag contradiction. *Journal of Computer Security*, 28(2): 269–293, 2020.

[240] L. Xu, G. Li, C. Li, W. Sun, W. Chen, and Z. Wang. Condroid: a container-based virtualization solution adapted for android devices. In *2015 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*, pages 81–88. IEEE, 2015.

[241] Z. Xu and S. Zhu. Abusing notification services on smartphones for phishing and spamming. In *WOOT*, pages 1–11, 2012.

[242] L. Xue, Y. Zhou, T. Chen, X. Luo, and G. Gu. Malton: Towards on-device non-invasive mobile malware analysis for {ART}. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 289–306, 2017.

[243] L.-K. Yan and H. Yin. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *USENIX security symposium*, pages 569–584, 2012.

[244] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *2015 IEEE/ACM 37th*

IEEE International Conference on Software Engineering, volume 1, pages 303–313. IEEE, 2015.

[245] C. Youmeng. Umeng. https://www.umeng.com/, 2023. Accessed online: January 26, 2024.

[246] Q. Zeng, L. Luo, Z. Qian, X. Du, and Z. Li. Resilient decentralized android application repackaging detection using logic bombs. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 50–61, 2018.

[247] L. Zhang, Z. Yang, Y. He, M. Li, S. Yang, M. Yang, Y. Zhang, and Z. Qian. App in the middle: Demystify application virtualization in android and its security threats. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(1): 1–24, 2019.

[248] N. Zhang, K. Yuan, M. Naveed, X. Zhou, and X. Wang. Leave me alone: App-level protection against runtime information gathering on android. In *2015 IEEE Symposium on Security and Privacy*, pages 915–930. IEEE, 2015.

[249] C. Zheng, T. Luo, Z. Xu, W. Hu, and X. Ouyang. Android plugin becomes a catastrophe to android ecosystem. In *Proceedings of the First Workshop on Radical and Experiential Security*, pages 61–64, 2018.

[250] W. Zhou, Z. Wang, Y. Zhou, and X. Jiang. Divilar: Diversifying intermediate language for anti-repackaging on android platform. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, pages 199–210, 2014.

[251] Zimperium. Grifthorse android trojan steals millions from over 10 million victims globally. https://www.zimperium.com/blog/grifthorse-android-trojan-steals-millions-from-over-10-million-victims-globally/, 2021. Accessed online: January 26, 2024.

Appendix A

Hash of the Samples

Due to the easy of readability, this thesis never reported the whole sha256, but only the first 4 bytes. For each sample, Table A.1 reports the full hash with the first 4 bytes in bold, sorted in alphabetical order. Moreover, each entry is a hyperlink to the corresponding VirusTotal webpage.

Table A.1 Sha256 of the samples mentioned in the thesis

0009	b9d191236fef80c954feb2eeea998d8c2a1e2ca6dc273a0a68836851423f
019e	12c7233d7324667d9e49aba4787c67204c5c8f2c38754f469a5b600bddde
0259	d084a78ddc98e663ae5799898a0afb4d021c6486cb61bdf7285731476d61
04ce	f547b64e459936dd243bfb19575bc905f6271c94723788000088f9e7e278
05b4	c4dd8bf9f376c767330e649d725ad35c0c9c3b1b2dbbfab7f39e90c5bac4
1306	8932cd52ffa257fa35bba7860e618416f0d53eecd7650a7700607220d4c0
180f	897219b41b01441d3fd275699b9eb7e922000b7ce16f60152389cc978f2e
1ac7	fad8a64016e4fdc185f604365b2333cabe65b8083242ed5d41c92a25a9fb
1f26	7514222943779bdd642b9c7322a31a87d8f17790be4f31d59c2f4fade4d3
213c	997dc02dfc4e83e872243c9217c7481a18a386b4fd79c049a5e27dad97f0
25ca	a43d5d96069b1cb8b9a9d5b18bd858b8ca2c4c0960d7b69d38d8414f68f0
32cd	907d3343c44180294a7c279c2a5f139a6ee443cbf443eb2bd663bca37c6e
338c	09c3ded12f3d6fed78706b1505d7cdf696fd6ac32913d2f710502853ba94
3c0b	88d1b054e275c4fa8b3496030bab8395b8789d1d4599bbd7ef4a13fca2c5
4a7e	913d491f715bb00b37ad5b8802a00c919070486212e8d1d1a802f4bdf6bf
4e4b	e579cffdd690cef4bb0d779d66ede95cfd955eb27eb797e0704f59d61e6d
5264	52b1ecb5c5572cfd24c9cb8b7cbeef5321510244a5f8a160492552698cc9
58b3	4234bd375ac81753cb8cc793a60cf9f0a220383bf332d15ce51917488623
5bd3	e6f49aaab9e7fe566d92cceb9a5701a072426434de5bb2cdbc34a7d265f2
6c6e	eed1b91913db0d6232edb1979c67d6fb48ca3da4f83dc49fb565a4e5f4fe
7383	7b030f031d532741b7e84068aabed24e7a6ac118c4272005e6ecd18a17d7
775c	3e036f3a1fc18fd683fa9e5da2a2e68f19feb7e7a0f609ba775a0a2e6571
7900	9a3bbdb9f73faa3d8b3a35306957fbd2bfb362d0c2d658079ff6a49b69e0
7adb	a016acdcafac4b5fa2eb44e3103b5cd80be16ae483d2008c72f79a22e0ac
7f9d	ce517a39bca41752f2deb028ea02b5408d0892133ec815e044354e95ceed
84a2	aabef11c823d55529f6424155dbf1f86ec32b601519457f79989cd992b1b
858e	bffaa54e40cc4787280da60e5854e8776359340bdf5287e32a580878a2c0
8cda	6e90ae30175dcfce5fca040abb525df4d2d74e81f52bd83971297683348a
9195	30d756b8e759023585656f8ae91ed743cb83c7f6765ee7244a93a17c8e7d
97e0	e7da3bacf383150d7ea1b4fe9ea502fe84aa856f1f51355b71260c453084
b816	209838a43e77dba33ed8d574f66735d9b1a239a110e53a82fa62e0a35f40
c227	edef2d823059f261b2101a21c4deeda2ee016671ce06b28dde0297018550
cdde	49edda06e3856755e5b847892ee91fb3ac334595328b1a742d9b898992a7
d867	31c8fa5ed48f13fadbf761a0869697dd56bbf963028e57d35395cf217f74
dcb7	2f950cbcbb6a8661b80ff15f83627a2ad4c55fdb8a3fc44fa752a96b4c91
e088	1b869add4b86628abb53255990aabb5db2548b259ecb04d03834dcf54d38
ecd2	981d192282fd72ca82cd3c13bc04fe366a411ebe8f76d8303298ac541f7d
f7b9	06ec2ce1c39979092dbd220d0b9bf7fb770122c4de31e239935aa4763fea
fcd4	d6ecfea993400fa242890e344c696724a6ac14bda9a0b067b0d418ffe5d7

Appendix B

Native Suspicious Tags

Table B.1 details the list of suspicious tags divided by categories that we identify in Section 4.3. '<SYMBOL>' denotes that there is a specific tag for each suspicious library call family (Table 4.1). A TAG is made of the concatenations of its category and title with the symbol -.

Table B.1 List of suspicious tags used in the ML validation.
†: float computed as $\frac{\#\ of\ features}{total}$
§: boolean

Step	TAG Category	TAG Category Description	TAG Title	TAG Example
#0	J_NATIVE_METHODS	Presence/absence of native methods and the entry point from which they can be reached	NO_NATIVE_METHOD†	
			NO_REACHABLE†	
			APP_LIFECYCLE_EP†	
			ACTIVITY_LIFECYCLE_EP†	
			EXTERNAL_DEX†	
			SUSPICIOUS_INTENT§	
#1	J_LOAD_METHODS	Presence/absence of load methods and the entry point from which they can be reached	NO_LOAD_METHODS§	
			PATH_LOAD_METHOD†	
			APP_LIFECYCLE_EP†	
			ACTIVITY_LIFECYCLE_EP†	
			SUSPICIOUS_INTENT§	
			EXTERNAL_DEX†	
			NO_ELF_NAME†	
			ELF_IN_LIB_AND_NOT§	
#2	CODE_LOCATION	Code file in suspicious location or with extension name mismatch	ELF_IN_ARCHIVE§	
			DEX_EXT_MISMATCH§	
			ELF_EXT_MISMATCH§	
#4	REGISTERNATIVES	`RegisterNatives` callback	UNRESOLVED_METHODS§	
			MULTIPLE_PATH§	
			CLASS_NON_IN_APK§	
#6A	NR_FINDCLASS	Native Reflection: `FindClass` callback	ANDROID_MANAGER§	
			CONTEXT§	
			CLASSLOADER§	
			JAVA_REFLECTION§	`java.lang.reflect.Method`
			THREAD§	
			SYSTEM§	
			CRYPTO§	`javax.crypto.Cipher`
			APP_INFO§	`android.content.SharedPreferences`
			ZIP§	
			ANDROID_INTERNALS§	`android.app.LoadedApk`
			STACK_TRACE§	`java.lang.StackTraceElement`
			EXCEPTION§	
			PARTIAL_RESOLUTION†	
			NO_RESOLUTION†	
	NR_METHOD	Native Reflection: `GetMethodID` callback	WITH_DANGEROUS_PERM§	
			ANDROID_MANAGER§	
			CONTEXT§	`getSystemService`
			SENSIBLE_INFORMATION§	`getImei`
			CLASSLOADER§	`loadClass`
			JAVA_REFLECTION§	`getClass`
			THREAD§	
			PERMISSION§	
			STACK_TRACE§	`getStackTrace`
			PARTIAL_RESOLUTION†	
			NO_RESOLUTION†	
#6B	DYNAMIC_LOADING	The tags report the usage of library call to dynamically load and invoke exported functions of other libraries	<SYMBOL>§ (see Table 4.1)	`dlsym(fd, "chmod")`
			NO_RESOLUTION§	
			ANDROID_DVM_ART§	`libdvm.so`
#6C	SUSP_LIB_CALL	Suspicious library calls	<SYMBOL>§ (see Table 4.1)	`execve`
			CREATEJAVAVM§	`JNI_CreateJavaVM`
	LIB_CALL_SUSP_PARAM	Suspicious argument to the library calls	<SYMBOL>§ (see Table 4.1)	`open("/proc/version")` `open("/sys/devices")`
			CREATEJAVAVM§	`JNI_CreateJavaVM`
#6A #6C	STRING	Presence of meaningful strings	CLASSLOADER§	
			ANDROID_INTERNALS§	
			PROPERTIES§	`ro.product.cpu.abi`

Appendix C

Anti-Behavioral Techniques

Table C.1 reports the 97 unique evasive methods we collected during the analysis. For each method, it highlights if it belongs to the DET or IET category.

	Technique	Description	Goal TAG	Goal Description	Impl. TAG	Implementation Description	Example
	ROOT	Root detection	APPS	Detect if 'root' apps are installed on the device	APP_INFO†	Query the package manager with a specific 'root' package name	PackageManager. getPackageInfo("magisk")
					INST_APPS§	Retrieve the list of all installed apps on the device	PackageManager. getInstalledPackages()
					STORAGE†	Try to access to the external storage of other 'root' apps	
					CMD†	Execute command to find/execute the 'root' apps	popen("magisk")
			SU	Check the presence of the 'su' binary	FILE†	Access to well known super-user paths	open("/system/bin/su")
					CMD†	Execute command to find/execute the 'su' binary	popen("which su")
			BUSYBOX	Check the presence of the 'busybox' binary	FILE†	Access to well known busybox paths	
					CMD†	Execute command to find/execute the 'busybox' binary	
			PROPS	Check root-related Android system properties	PROP†	Retrieve a specific system property	
					CMD†	Retrieve a specific system property through command line	popen ("getprop ro.build.tags")
					FOREACH§	Retrieve all Android system property	popen("getprop")
			RO_PATHS	Check if some paths that should be only readable is also writable	FILE§	Check the property of the 'read-only' paths	faccess("/system/bin")
					CMD§	Check the mounted partitions	popen("mount")
	DEBUG	Debugging detection	PROPS	Check debug-related Android system properies	PROP†	Retrieve a specific system property	ro.debuggable
					CMD†	Retrieve a specific system property through command line	
					FOREACH§	Retrieve all Android system property	
			MANIFEST	Check if the app was built with the debugabble flag enable	APP_INFO§	Query the package manager to retrieve the app metadata with the GET_ATTRIBUTIONS flag	
					INST_APPS§	Retrieve the info related to the GET_ATTRIBUTIONS flag for all installed apps	
			IS_CON	Check if a debugger is connected	API†	android.os.Debug. isDebuggerConnected()	

		JDWP	Check the presence of the JDWP debugger	FILE§	Parse the 'comm' files under the '/proc/self/*' directory	open ("proc/self/comm")
				CMD§		popen ("cat /proc/self/comm")
		PORT	Check if the default debugger port is already taken	FILE§	Parse the /proc/net/tcp file	open("proc/net/tcp")
				CMD§		popen ("cat /proc/net/tcp")
				SOCKET†	Connect with a socket to the default debugger port	socket ('127.0.0.1', 23946)
		TRACERPID	Check for the presence of a tracer PID	FILE§	Parse the status file of the current process or the tasks of the process	
				CMD§		
		TIME	Time checks	API§	Check the elapsed time in Java or C/C++ code	SystemClock. elapsedRealtime()
				FILE§	Parse the '/proc/utime' to retrive the elapsed time	
				CMD§		
HOOK	Hook detection	APPS	Detect if 'hook' apps are installed on the device	APP_INFO†	Query the package manager with a specific 'hook' package name	PackageManager. getPackageInfo("xposed")
				INST_APPS§	Retrieve the list of all installed apps on the device	
				STORAGE†	Try to access to the external storage of other 'hook' apps	
				CMD†	Execute command to find/execute the 'hook' apps	popen ("find . -name *xposed*")
		PROC_ART	Detect artifacts in the process memory mapping	MAPS§	Parse the '/proc/self/maps' file	
				NF§	Inspect the memory mapping of the process through C/C++ functions	
				CMD§		popen("ps")
				API§	Check high-level process info by quering the Android APIs	Application. getProcessName()
		STACKTRACE	Retrieve and parse the stacktrace to detect call graph artifacts	API§		Thread. getStackTrace()
		FRIDA	Frida-specific controls (not contained in previous checks)	FILE§	Check the content of the '/proc/self/fs/*' files	
				CMD§		

				SOCKET†	Connect to well known ip/port of the frameworks	socket ("127.0.0.1", 27042)
		CLASS	Detect if the target app retrives well known framework classes	API†		de.robv.android.*
EMU	Emulator detection	PROPS	Check emulator-related Android system properties	PROP†	Access to a specific system property	
				CMD†	Retrieve a specific system property through command line	
				FOREACH§	Retrieve all Android system property	
		ADB	Check if the adb is emulated	FILE§	Check the content of the '/proc/net/tcp' file	
				CMD§		
		SENSOR	Check if sensors' values are not related to real behavior	API§	Register a listener to collect sensors' data	gyroscope
		SYSTEM	Check System related properties (e.g., Device ID, Subscriber ID)	API†		TelephonyManager. getLine1Number()
				STATS§	Get (file-)system stats	ustat
				LOGCAT§	Exploit *logcat* to retrieve system information	
		KNOWN_EMU	Check artifact of well known emulators' artifacts	FILE†	Check the presence of the file	access ("/dev/socket/genyd")
				CMD†		popen ("ls /init.vbox86.rc")
		QEMU	Check QEMU artifacts	FILE†	Check the presence of a file	open ("/dev/qemu_pipe")
				PROP†	Access to QEMU-specific system properties	
				PROC§	Check hardware informations	/proc/cpuinfo
				CMD§		
		BEHAVIOR	Check (emulated) device features	BATTERY§	Check the battery status through API or broadcast receivers	New receiver for: BATTERY_CHANGED
				USER_PROF§		
				HARDWARE§	Check hardware-related discrepancy. For instance, graphical low video frame rate	
MEMTMP	Memory integrity verification	MEM2DISK	Check the content of a file w.r.t. the one loaded in memory	FILE§	Open libraries that are already mapped into the memory of the process	
		PLT	Check for PLT modifications	—	It is not possible to detect	
		INLINE	Check for inline breakpoints or trampolines	—	It is not possible to detect	

VIRT	App-level virtualization detection	FAKE_COMP	Check components' name w.r.t. the ones in the Manifest	API[§]	Retrieve the list of registered/running components of the app	
				APP_INFO[§]	Query the package manager to retrieve the metadata of the app with one of the following flags: GET_ACTIVITIES, GET_ATTRIBUTIONS, GET_RECEIVERS, GET_SERVICES, GET_PROVIDERS, GET_SHARED_LIBRARY_FILES	
				INST_APPS[§]	Retrieve the info for all installed apps with one of the following flags: GET_ACTIVITIES, GET_ATTRIBUTIONS, GET_RECEIVERS, GET_SERVICES, GET_PROVIDERS, GET_SHARED_LIBRARY_FILES	
		APP	Check if the app is really installed on the device	APP_INFO[§]	Query the package manager to retrieve the metadata of the app itself	
				INST_APPS[§]	Retrieve the info for all installed apps	
				NATIVE[§]	Retrieve the info of native libraries	
		PROC	Check the app's processes	CMD[§]		popen("ps")
				API[§]	Retrieve the list of running components of the app and their metadata	Application. getProcessName()
		UND_PERMS	Check if the app has or check for permissions that are not declared in the Manifest file	API[†]		Context. checkPermission (<no_manifest_perm>)
				APP_INFO[§]	Query the package manager to retrieve the metadata of the app with the GET_PERMISSIONS, or GET_URI_PERMISSIONS_PATTERNS flag	
				INST_APPS[§]	Retrieve the info for all installed apps with GET_PERMISSIONS, or GET_URI_PERMISSIONS_PATTERNS flag	

		APP_DIR	Check the installation path and other app's private folders	APP_INFO[§]	Query the package manager to retrieve the metadata of the app with the GET_METADATA flag	
				INST_APPS[§]	Retrieve the info for all installed apps with the GET_METADATA flag	
NET	Network detection	ADB	Check if the adb is emulated	SOCKET[†]	Connect to the default adb port	socket ("127.0.0.1", 5555)
		VPN	Check the presence of a VPN	API[†]		NetworkCapabilities. hasTransport()
				APP_INFO[†]	Query the package manager to retrieve the metadata of well known VPN apps	
				INST_APPS[§]	Retrieve the info for all installed apps	
				STORAGE[†]	Try to access to the external storage of other VPN apps	
				CMD[†]	Execute command to find/execute the VPN apps	
		INTERFACE	Check the metadata of the network interfaces	NF[§]		getifaddrs()
				FILE[§]	Check the content of '/proc/sys/net/ipv(4|6)' and '/sys/class/net'	
				CMD[§]	Command line commnds	popen("ifconfig")
				API[§]		ConnectivityManager. getNetworkInfo()
		FAKEIP	Check if the current IP is not real	CMD[§]		/system/bin/netcfg
		KNOWNIP	Check if the current IP is well known	SOCKET[†]		maxmind.com
		ADB	Check if the standard adb port is already used	SOCKET[†]		maxmind.com
		SSL_PINNING	Check if the app uses ssl pinning	API[§]	The app uses the SSL Context to perform a network request	
		LISTENER	Monitor changes for network	API[§]	Register a new listener though the Android API	ConnectivityManager. listenForNetwork()
				BR[§]	Register a broadcast receiver at runtime for: 'CONNECTIVITY_CHANGE', 'WIFI_STATE_CHANGED', and 'AIRPLANE_MODE'	

APK Tampering Verification	SIGNATURE	Signature check	ZIP	Read the certificate files in the APK	FILE§	Access to the APK	ZipFile ("/path/to/base.apk")
					CMD§		popen ("unzip base.apk")
			APP		APP_INFO§	Query the package manager to retrieve the metadata of the app with GET_SIGNATURES, or GET_SIGNING_CERTIFICATES flags	
					INST_APPS§	Retrieve the info for all installed apps with the GET_SIGNATURES, or GET_SIGNING_CERTIFICATES flags	
	INSTALL	Installer Verification	SOURCE		API†		PackageManager. getInstallSourceInfo()
High-level Verification	GOOGLE	SafetyNet & Integrity API	SN	SafetyNet checks	BINDER†	Check the binder methods for the SafetyNet requests	
			IA	Integrity API	API†		
	HUMAN	Interaction with a human being	—		—	It is not possible to detect	

Table C.1 List of evasive techniques.
†: Direct Evasive Technique (DET)
§: Indirect Evasive Technique (IET)