Demystifying Anti-Repackaging on Android

Antonio Ruggia, Luigi Sciolla

Department of Computer Science, Bioengineering, Robotics and System Engineering (DIBRIS)

University of Genova

Supervisor

Alessio Merlo, Luca Verderame

In partial fulfillment of the requirements for the degree of

Master in Computer Engineering - Software and Computing Platforms

December 22, 2020

Acknowledgements

I miei più sinceri ringraziamenti vanno al mio relatore, Prof. Alessio Merlo, che mi ha seguito, giorno dopo giorno, in questo percorso. Grazie per la sua disponibilità in tutti questi mesi, a qualsiasi ora del giorno e della notte. Grazie al mio correlatore, Prof. Luca Verderame, con la sua competenza sull'argomento ci ha aiutati durante lo sviluppo e la stesura di questo elaborato.

Ringrazio il mio collega e amico Antonio Ruggia, con cui ho condiviso nell'ultimo anno giornate di lavoro e divertimento. Sono orgoglioso dei risultati che abbiamo raggiunto e delle conoscenze che abbiamo acquisito insieme.

Grazie a tutti i miei quattro nonni: tutti mi hanno passato qualcosa, la spinta educativa, morale e intellettuale (nonché quella fisica, con tutti i pasti cucinati con amore) per arrivare a questo risultato.

Grazie ai miei genitori, a mia madre per l'enorme supporto, anche economico, in questa mia scelta accademica. E a mio padre, per avermi supportato e dato motivi di distrazione nei momenti più difficili.

Un pensiero di ringraziamento va a Gaia, più di tutti mi è stata vicina in questi anni, dandomi i giusti tempi e spazi. Ha condiviso con me ogni soddisfazione e delusione. Sei stata speciale.

Un ringraziamento a tutte quelle persone senza le quali non sarei arrivato a scrivere questa tesi. I compagni delle superiori che hanno

sopportato i miei discorsi: Camilla, Filippo, Lorenzi, Simone, Walter. Tutti i colleghi di questi anni, non scorderò le mattinate in villa Imperiale, i pomeriggi dagli edili, i pranzi sulla panchina, le cene con amici in casa di Andre, fino ad arrivre alle chiamate su zoom: in particolare Andrea, Emanuele, Eugenio, Filippo, Giacomo, Lorenzo e Marco. Tutto il team ZenHack che ha provato più e più volte di distrarmi dalla laurea, ma che mi hanno passato conoscenze incalcolabili: Andrea, Emilio, Enrico, Francesco, Gaspare, Giotino, Giovanni, Lorenzo, Luca, Marina, Simone, Tiziano e tutti i nuovi .

Grazie ad una persona che negli ultimi mesi mi ha dato la spinta per finire: Ago. Sei stato sorprendente: nonostante il periodo difficile sei riuscito a dare forza a te stesso e a tutti quelli che ti sono vicino.

L'ultimo grazie alla mia mente per il suo aiuto nei miei studi, e che sicuramente si sarà dimenticata di scrivere qui qualche persona che è stata importante per me in questo percorso, magari più di quelle sopra citate.

Abstract

App repackaging refers to the practice of customizing an existing mobile app and redistributing it in the wild. In this way, the attacker aims to force some mobile users to install the repackaged (likely malicious) app instead of the original one. This phenomenon strongly affects Android, where apps are available on public stores, and the only requirement for an app to execute properly is to be digitally signed.

Anti-repackaging techniques try counteracting this attack by adding logical controls in the app at compile-time. Such controls activate in case of repackaging and lead the repackaged app to fail at runtime. On the other side, the attacker must detect and bypass the controls to repackage safely. The high-availability of working repackaged apps in the Android ecosystem suggests that the attacker's side is winning.

In this respect, this thesis aims at bringing out the main issues of the current approaches to anti-repackaging and propose a brand new anti-repackaging software which exceeds the limits of previous solutions.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

During the last five years, Android strengthened its leadership among mobile operating systems with more than 75% of market share on average [1]. Most of this success is due to the app development process, which is easier compared to other mobile operating systems: in fact, Google currently offers a plethora of platforms that support the developer in designing, implementing, testing and sharing her application [2]. As a consequence, the number of Android apps in the Google Play Store reached 2.8M in 2020[1].

From a security standpoint, Android apps are the main target for attackers, as they allow to reach a large number of mobile users. In fact, since a mobile app is available in public app markets (e.g., Google Play Store, Samsung App Store, ...), an attacker can easily retrieve the app bundle (i.e., the *apk* file, which is basically an archive containing the app). Then, the attacker can reverse engineer the *apk* [3], inject some malicious code, and re-distribute a modified version of the original app. This kind of attack is called *repackaging*.

Android repackaging is a serious problem that may affect any unprotected app. A repackaged app can cause money losses for developers as the attacker

[1] https://www.appbrain.com/stats/number-of-android-apps

can re-distribute paid apps for free, remove/redirect ads earnings, or make the app willingly unusable to negatively impact the reputation of the developer. Furthermore, recent studies [4] demonstrated that the 86% of Android malware is contained in repackaged apps. This is indeed a promising strategy for malware developers, as embedding the malicious code in a repackaged version of popular and well-ranked app speeds up its spread.

It is worth pointing out that repackaging is intrinsically related to the Android development life-cycle, where an *apk* just needs to have a valid signature in order to be successfully installed and execute properly, without requiring any guarantee on the actual identity of the signer (i.e., the developer does not need a valid public key certificate issued by a trusted certificate authority). Therefore, an attacker can modify an existing *apk* and then repackage and sign it with her own self-generated private key.

In order to make the repackaging phase more challenging for attackers, in recent years several techniques have been proposed in literature. Such techniques can be divided into two sets, namely *repackaging detection* and *anti-repackaging*: the first set aims at recognizing repackaged apps, while the latter one focuses on inserting proper controls in the app, such that a repackaged version of the same app would not work properly.

While repackaging detection techniques have been widely discussed and analyzed in literature [5; 6], no systematic analysis of anti-repackaging techniques is still available. To this aim, in this thesis we provide an extensive analysis of such techniques. It is two-fold: the first part, until 3 will analyze the state of the art; starting from 4 we will develop a brand new anti repackaging tool.

More in details, the first part is subdivided in three part: after introducing some background and the threat model related to app repackaging in Chapter 2, we first discuss the state of the art of anti-repackaging techniques in Chap-

ter 3. Then, in Chapter 4.1 we summarize the attacking techniques that can be used to circumvent anti-repackaging. In Chapter 4.2 we will show how existing anti-repackaging techniques can be circumvented to produce a fully-working repackaged version of a protected app. We also discuss a full-fledged attack to the most recent anti-repackaging technique to date, which is the only one whose source code is publicly available. More in detail, we show how to bypass the protections and successfully repackage a real app (i.e., Antimine [7]). In Chapter 5 we point out some guidelines to improve the robustness and the reliability of anti-repackaging techniques and we propose a novel approach that implements these improvements. Finally, in the last Chapter we draw some conclusions and future works.

Chapter 2

App Repackaging: Backgroud and Threat Model

Summary

In this chapter we briefly recap the basics of app development and distribution in Android. Then, we discuss a threat model for app repackaging, and we summarizes the main categories of repackaging detection and anti-repackaging techniques.

2.1 Android App life-cycle

App development. A native Android app is developed in Java or Kotlin, and leverages XML files to represent the graphical user interface of the app. From now on, we will explicitly refer to Java code, although the same techniques are valid also in case of Kotlin-based apps. There are other ways to develop Android apps, for example hybrid apps. Hybrid apps are mainly built using web technologies

and run over a WebView (e.g., Apache Cordoba[1]), and are likewise weak against repackaging.

Each Android app is distributed and installed as an Android Package (*apk*) file. Until 2018, developers had to build their *apk* file to distribute their app. Since 2018 Google provides a new publishing format, i.e., the Android Application Bundle (AAB) [8], which allows building optimized *apk* for each device configuration[2].

In a nutshell, an *apk* file is a zip archive containing all the necessary files to run the first execution of the app. Each *apk* is signed with its developer's private key and also contains the corresponding public certificate of the developer. This mechanism grants the integrity of the *apk*, but cannot provide any authentication, as the developer certificate does not need to be issued by a trusted certificate authority. Therefore, Android performs only basic 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.

Regarding app repackaging, the most relevant files and folders contained in the *apk* are:

- *META-INF/*: it contains the developer's public certificate and the app signature;
- *lib/*: it contains the native libraries for each specific architecture (e.g., x86, armeabiv7a). An app could excute native C/C++ code through the Java Native Interface (JNI)[3];
- *res/*: it contains the binary resources, such as images or hybrid apps content;

[1] https://cordova.apache.org/

[2] Hereafter we will refer to *apk* only, as the distinction between AAB and *apk* is irrelevant for the problem at stake.

[3] https://developer.android.com/ndk

- *AndroidManifest.xml*: it describes the structure of the app and its components. In this file the developer defines app permissions, entry points, intent filters, and more;

- *classes.dex*: it contains the compiled Java/Kotlin code in the form of bytecode, i.e., a high level representation of the machine code, that could be converted into Smali code, i.e., a human readable format, with off-the-shelf tools like Backsmali [9].

Application distribution. Android apps are mainly delivered through proper software repositories, called *app stores*, which allow developers to publish their *apk*. iOS relies on a single, centralized market (i.e., the Apple Store), which is directly controlled by Apple. On the contrary, a distinctive feature of Android is the possibility to distribute the same app on dozens of different app stores beyond the official one (i.e., the Google Play Store[1]). The most popular app stores differ from region to region. In Europe and US, the Google Play Store has the greatest market share [10]. However, in 2019, in China, *Tencent My App* was the biggest Android app store with up to 25.5% of the market share, followed by the *360 Mobile Assistant* and *Xiaomi App Store*; in this country the Google Play Store is just tenth place [10; 11].

Whenever a user downloads an app from any store, Android carries out a soft check on the app certificate before installation, with the final decision left to the user [12]: we recall that an *apk* has to be signed by the developer, but Android continues the installation even if the signer of the developer's certificate is unknown [13]. Since the only constraint is a valid signature, it is also possible to distribute the *apk* directly (e.g., through email or urls) without the need to upload the app on an app store.

[1] `https://play.google.com/store`

2.2 App Repackaging: Threat model

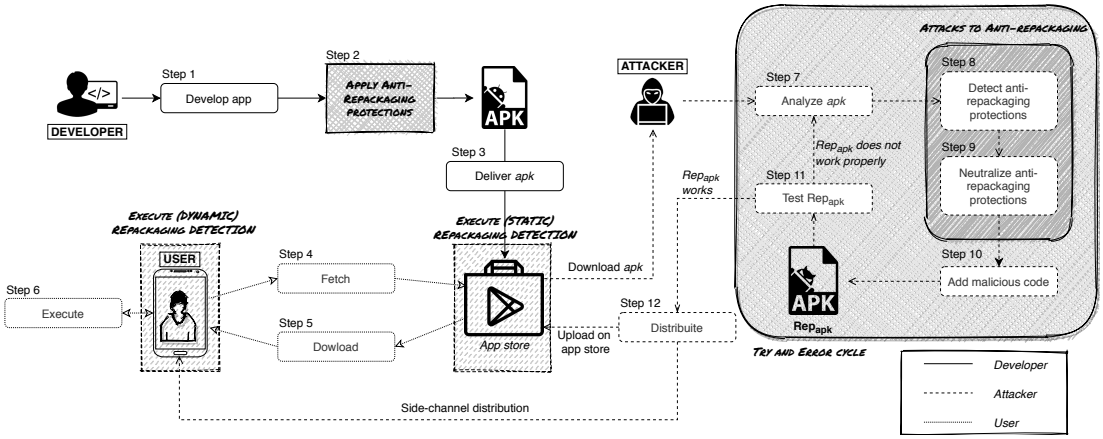


Figure 2.1: App repackaging: Threat model.

In this section we discuss a threat model for app repackaging. The model is depicted in Fig. 2.1.

The threat model involves three actors, namely the developer, the attacker and the user. The developer implements the app (Step 1), and may also add some anti-repackaging protection in the code before building the *apk* (Step 2), in order to counteract repackaging attempts. In general, applying an anti-repackaging technique means to add some extra code that may lead, for instance, the repackaged app to crash. Then, the developer builds and signs the *apk* and delivers it to the app store (Step 3).

In an ideal (i.e., attack-free) scenario, the user fetches the store for apps (Step 4), then chooses and downloads the original *apk*. The validity of the signature and the integrity of the *apk* is verified automatically by Android, by leveraging the developer's public key certificate provided in the *apk*. Then, the *apk* is installed successfully and the user can execute the app (Step 6).

In an actual use case, an attacker can download the developer's *apk* by behaving as any normal user. Then, the attacker decompiles and analyzes the *apk*

(Step 7), and tries to detect the anti-repackaging protections (Step 8). More in details, the attacker can decompile the *apk* using Apktool [14] and perform static analysis on the smali source code. Smali is a human-readable format that can be obtained from the Dalvik bytecode contained in the *classes.dex* file. Most of the smali code could be further decompiled back into Java code [15]. If static analysis is not sufficient, an attacker can dynamically infer the app behavior at runtime, by relying, for instance, on Frida [16]. A fruitful combination of static and dynamic analysis can allow the attacker to detect the anti-repackaging protections added by the developer in the app code. In case some protections are detected, the attacker tries to neutralize them (Step 9) by deactivating the corresponding extra code, accordingly. Then, he can add some malicious code (Step 10) and build and signs a repackaged version of the *apk* (i.e., Rep_{apk}) with a valid self-generated public key certificate. Finally, the attacker tests Rep_{apk} on her own devices (Step 11). In case it works, the attacker can try distributing Rep_{apk} (Step 12) both on official channels (i.e., app stores) and side-channels (e.g., by email, on website, through phishing attacks, ...). Otherwise, the attacker needs to carry out further analysis on the *apk*, and repeat Steps 7 to 11 (also known as the *try and error cycle*) until she obtains a working Rep_{apk}.

The attack succeeds whether some users install Rep_{apk} instead of the original *apk*. In this case, the execution of Rep_{apk} leads the user to execute the malicious code on her own device. The malicious code may have different aims, like e.g., access the app premium features for free or redirect revenue to cause financial loss to the developer, gain access to the user's private in app data (e.g., credentials), or make the app willingly unstable to affect the reputation of the original app.

2.3 Repackaging Detection vs. Anti-Repackaging

This section points out the main features of both *repackaging detection* and *anti-repackaging* techniques.

2.3.1 Repackaging Detection

Repackaging detection focuses on recognizing repackaged apps in app stores or on user's devices, with the aim to limit the spread of repackaged app (i.e., Step 12 in Figure 2.1). Repackaging detection techniques can be divided into static and dynamic approaches [17], and further grouped into five different categories (see [6]).

- **Static or offline techniques** are adopted on *app stores* to identify uploads of repackaged apps which are clones of original apps. Such techniques strongly relies on *symptom discovery* or *app similarity*. Symptoms [6] are tracks, such as strings, left by the repackaging process itself. Similarity checks usually involve a two-step process in which each app is first profiled according to a set of distinctive features; then, apps with almost identical feature profiles are identified. Such techniques make strong use of machine learning, both supervised [18; 19] and unsupervised [20]. Static analysis needs a considerable amount of resources both in order to generate the decision model and to make the final decision, and can be efficiently applied only on server side.

- **Dynamic or online techniques** are directly executed on the user's device to evaluate the app. These approaches extract specific information both at install-time and at run-time. For instance, the technique discussed in [21] puts a watermark inside the app which is checked by a third authority at runtime.

Attacking Repackaging Detection. An attacker successfully circumvents a repackaging detection technique once it is able to build repackaged apps that go undetected both on the app store and on the devices. According to the characteristics of the current detection techniques, this means that the repackaged app must appear very different from the original app (to bypass similarity checks), hide repackaging identifiers (to avoid symptom discovery), and behave as the original app to circumvent dynamic analysis. It is also worth pointing out that the granularity of the Android ecosystem - which allows to deliver apps through an undefined number of app stores and side channels (e.g., apps available at some URIs or delivered by email) - makes hard to build up a reliable, pervasive and full-fledged repackaging detection deployment. In fact, this would require to deploy static analysis mechanisms on each source of apps (i.e., app stores and servers), as well as dynamic analysis solutions on each mobile device.

2.3.2 Anti-Repackaging

Anti-Repackaging - also known as *repackaging avoidance* or *self-protection* - aims to protect an app from being successfully repackaged. As opposed to repackaging detection, anti-repackaging is applied by the developer to the app code before building the *apk* (Step 2) and aims to make any repackaged version of the same app crash. On the bad side, the attacker aims to detect and neutralize the anti-repackaging protection after reversing the *apk* (Steps 8 and 9). From a technical standpoint, anti-repackaging techniques insert some logic controls (i.e., *anti-tampering* checks) inside the *apk*. Since the first aim of the attacker is to recognize anti-repackaging, the requirements of anti-tampering checks are i) to go undetected and hide inside the app code, and ii) in case of detection, to avoid being de-activated. To achieve such results, anti-tampering checks may rely on code obfuscation, code virtualization and anti-debugging techniques. We briefly

introduce anti-tampering checks and such techniques.

Anti-tampering. In a nutshell, anti-tampering checks are self-protecting functions which aim to detect any modification on the original app. There exist several methods to detect tampering in mobile apps, as highlighted in [22]. The most relevant anti-tampering techniques are:

- **Signature checking**: the most trivial control, which checks the certificate contained in the *apk*. This approach could detect any modification applied to the original *apk*;
- **Code integrity**: it checks whether some specific part of the code has been tampered with, by computing its signature at run-time. As an example, the first bytes of the *.dex* file contains the hash of its content. This method detects a tampering only if the attacker has modified the checked methods;
- **Resource integrity**: it checks the signature of some resources at run-time. This technique differs from the previous ones as it checks tampering attempts in the app resources (e.g., images and other binary files) instead of the code;
- **Installer verification**: it checks if the installing app comes from trusted app stores. This technique is applied under the assumption that tampered apps are more likely to be distributed on unofficial app stores.

Anti-tampering checks dynamically react to any modification either in the app or in the executing environment. Once some checks are detected by the attacker (Step 8 in Figure 2.1), the same attacker tries to bypass them by following the *try and error* approach, i.e., she modifies the app (in order to try removing checks), then she repackages and tests the app to check whether it works properly. If the

attempt fails, then the attacker may try to search for other checks and/or modify the app in a different way.

Code obfuscation. Obfuscation approaches modify the bytecode or the source code of an app, without changing its behavior, in order to make some manual or automatic analysis more difficult. There exist a lot of off-the-shelf obfuscation tools both open source [23] and commercial [24].

Code virtualization. Code virtualization is a specific obfuscation scheme. In this approach, instructions are mapped to semantically-equivalent virtual instructions. A virtual machine (VM) executes inside the app and interprets the virtual instructions. Two different approaches are proposed in [25; 26]; the first one works at DEX level, while the latter focuses on native libraries. Similarly to obfuscation, this approach aims at complicating the reverse engineering process of an app, but it does not introduce any anti-tampering check.

Anti-debugging. Anti-debugging controls aim to detect and avoid dynamic analysis checks (e.g., debugger). Differently from passive obfuscation techniques, where the app code is syntactically modified to make it harder to understand, anti-debugging controls allow an app to actively react against malicious reverse engineering at run-time [22]. The most widespread anti-debugging techniques are:

- **Emulator detection**: it checks whether an application is running on a real device or in an emulated environment;
- **Time check**: when an app executes with a debugger, its execution slows down. Time checks can be specific functions whose execution depend on the

execution time: in case the execution time is higher than a given threshold, such functions take the app to an inconsistent state;

- **Check debuggable**: the *AndroidManifest* contains a flag that enables the debugging mode: if this flag is set to false, an user is not able to attach any debugger to the application. The attacker can bypass this check by changing the manifest accordingly. Nonetheless, such modification could trigger some other anti-tampering checks.

Recently, more complex anti-debugging techniques have been put forward [27]. Among these, the most promising one is discussed in [28]: the intuition is to exploit proper features of the Android Runtime to hide anti-debugging instruction at runtime. As any other solution relying on version-specific features of the Android OS, also this approach quickly became obsolete, as it worked properly only up to API 25 (i.e., Android v. 7.1).

Chapter 3

Anti-Repackaging: State of the art

Summary

This chapter presents the state of the art of anti-repackaging techniques on Android, ordered chronologically. For each proposal, we will describe the idea, the protection scheme applied on the app before release, the runtime protection during the app execution, as well as the experimental results. In Chapter 4.2 we will discuss limitations and effective attacks against these works.

3.1 Self-Protection through `dex` Encryption

In 2015, M. Protsenko et al. in [29] proposed an anti-repackaging technique aimed at complicating both the reverse engineering and the repackaging process. The main idea is to encrypt the `classes.dex` file in the *apk*, and dynamically decrypt and execute it at runtime.

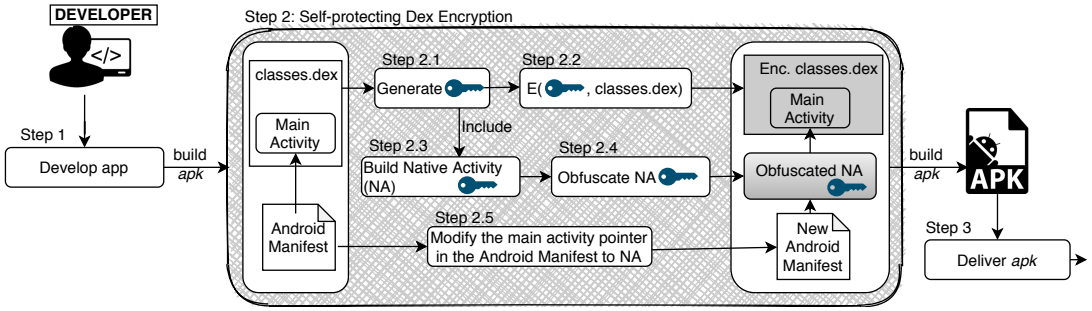


Figure 3.1: The Dynamic Self-Protection and Tamperproofing scheme.

Protection scheme: The approach, depicted in Fig. 3.1, applies to the compiled *apk*, that is modified according to the following steps:

1. A XOR key is generated (Step 2.1) and applied to encrypt the whole `classes.dex` file(s) (Step 2.2);
2. An native activity[1] is automatically built (Step 2.3) and compiled in a shared library. Such activity is called native, as it is written in native code (C/C++ code) instead of Java/Kotlin code as the other activities. The XOR key is added to the native activity;
3. The native activity is obfuscated using the OLLVM tool [30] (Step 2.4);
4. The Android Manifest is configured to execute the native activity (written in C/C++) rather than the original main activity[2] when the app starts executing (Step 2.5).

Finally, the *apk* file is generated, signed and delivered (Step 3).

Runtime behavior: When the app executes, the protection scheme behaves as depicted in Fig. 3.2. At first, the modified manifest forces the execution of the native activity (Step 6.1). The native activity begins by extracting the XOR key

[1]An activity is an app window containing the UI.
[2]The Main Activity is the first activity prompted to the user when the app is executed.

(Step 6.2) and launches the main activity (step 6.3). As the execution of the main activity requires access to the plain bytecode, the XOR key is applied to decrypt only the part of the code required for the proper execution of the main activity. Then, after the execution is completed, the same bytecode is re-encrypted using the same key (Step 6.4). The approach is time-sliced, thereby decrypting parts of the `classes.dex` file on-demand, i.e., maintaining the bytecode unencrypted only when it needs to execute (Step 6.4).

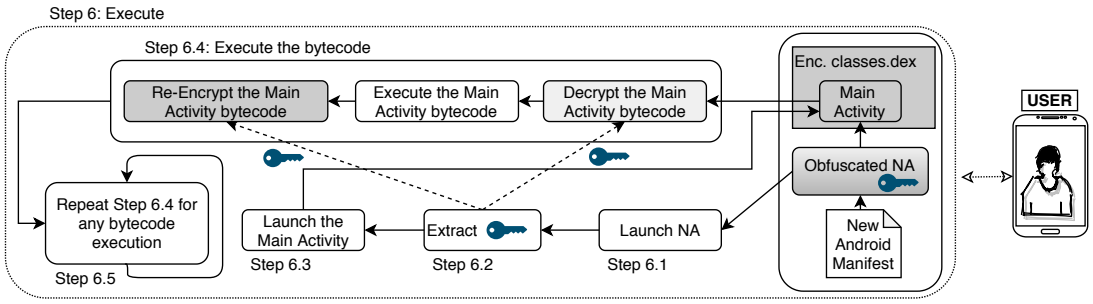


Figure 3.2: Dynamic Self-Protection and Tamperproofing: runtime behavior

Evaluation: The adoption of a XOR-based encryption of the bytecode is aimed at implementing a tamper-proof checksum. This work has been evaluated on 749 apps from the F-Droid app store [31]. Due to implementation and process choices, only 312 apps can be properly protected. For instance, it was not possible to protect apps which make use of Java reflection. The experimental setup leveraged several Android devices equipped with Android 4.4.2, and stimulated through the Android Monkey UI Exerciser framework [32] at runtime. Clearly, the adoption of encrypting/decrypting cycles, as well as the addition of extra code, negatively impacts the app time and space complexity. In fact, the experiments indicate a mean execution overhead of 440% and an increased *apk* size by 183% on average.

The reliability of the proposed protection scheme is evaluated according to four metrics: method exposure (both absolute and percent value), and instructions

exposure (both absolute and percent value). A method or a instruction is *exposed* when it is decrypted. Experimental results show a wide variance. On average, about 83% of methods - which amount to the 80% of the bytecode instructions - are exposed.

3.2 Stochastic Stealthy Networks

In 2016, Lannan Luo et al. put forward another self-protecting approach against repackaging, named Stochastic Stealthy Network (SSN) [33].

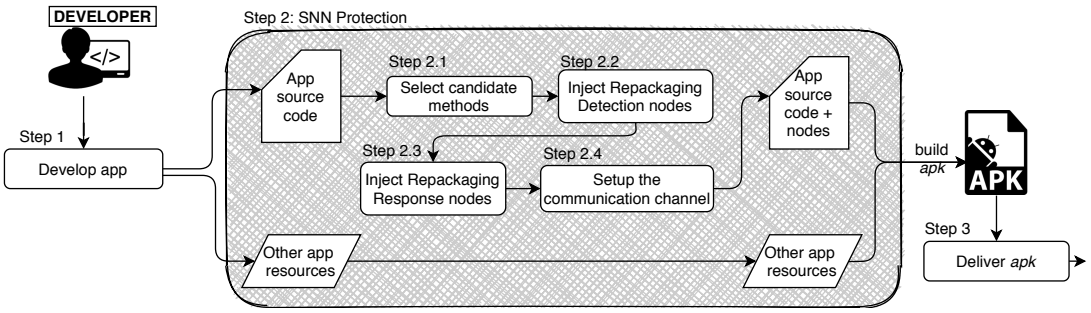


Figure 3.3: The SSN protection scheme

Protection scheme: The idea is to distribute (and hide) a set of guards (i.e., *repackaging detection* nodes) into the app source code. When executed, such guards trigger proper decision points (i.e., *repackaging response* nodes) that, in case a tampering is detected, make the app crash. More in detail, the protection workflow, depicted in Figure 3.3, begins by selecting a set of candidate methods in the Java code, according to some heuristics (Step 2.1). Then, a set of repackaging detection nodes are injected in such methods (Step 2.2), as well as the set of repackaging response nodes (Step 2.3). Such latter nodes take decisions according to a stochastic function. Moreover, a communication channel is added in the code to allow both kinds of node to communicate at runtime (Step 2.4). The

communication channel is made by extra variables and methods added in the app code. Finally, SSN builds, signs and delivers an *apk* with code extended with the repackaging nodes.

It is worth noticing that the selection of candidate methods must take into consideration the overhead due to the execution of detection and response nodes: to this aim, good candidate methods must be seldom invoked at runtime. Furthermore, the detection nodes must be strongly distributed over the set of candidate methods, to increase stealthiness.

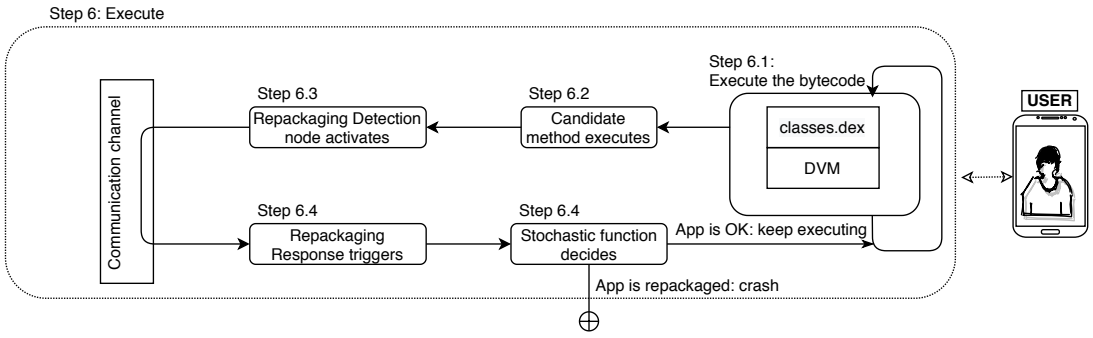


Figure 3.4: The SSN behavior at runtime.

Runtime behavior: Fig. 3.4 depicts the SSN behavior during app execution. Whenever a candidate method executes (Step 6.2) and activates a repackaging detection node (Step 6.3), then the same node connects to the corresponding response node(s) through the communication channel. If the stochastic function recognizes the method as repackaged (i.e., modified) then some execution is triggered. Such execution may lead to modify specific variable values or raise specific exceptions[1] that can result in program crashes or delayed logical malfunctions.

Evaluation: The SSN approach only verifies the integrity of candidate methods by relying on the developer's public key verification. However, SSN supports sev-

[1]Unfortunately, the paper does not provide such details.

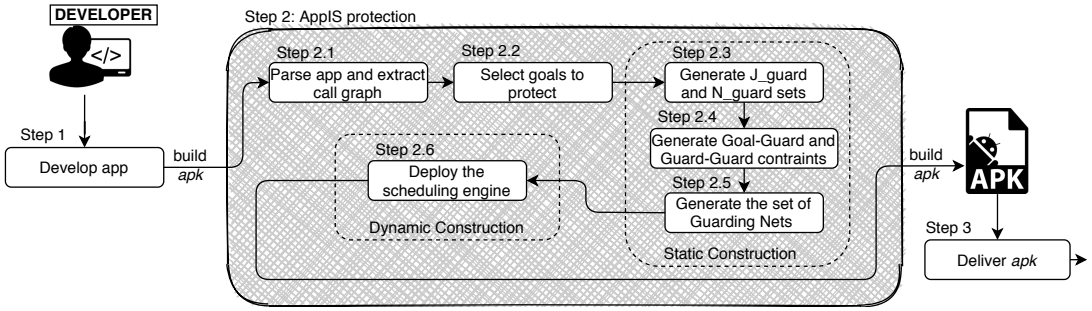


Figure 3.5: The AppIS protection scheme.

eral other predefined templates. For instance, reflection is used to call functions such as `getPublicKey()` and `generateCertificate()`.

The viability of the SSN approach has been empirically evaluated on a set of 600 apps, belonging to ten app categories in F-Droid. Experiments were conducted on emulators equipped with Android 4.1. The average time overhead is between 6.4% and 12.4%, which is far lower than the time overhead of the previous approach, discussed in Section 3.1. Furthermore, each protected app works properly, and no space overhead is noticed.

3.3 AppIS: Protect Android Apps Against Runtime Repackaging Attacks

In 2017, Song et al.[34] proposed an app reinforcing framework, named AppIS. The idea of AppIS is to add security units as guards with interlocking relationship between each other, in order to build redundant and reliable anti-repackaging checks.

Protection scheme: Fig. 3.5 depicts the protection workflow of AppIS. The AppIS workflow begins by parsing the compiled *apk* to extract its corresponding function call graph (Step 2.1), leveraging static analysis techniques at the state of

the art. From such graph, AppIS selects a set of sensitive methods, resources or data to protect (Step 2.2), named *goals*. Then, AppIS generates a set of guards aimed to protect goals (Step 2.3). Guards can be hosted both in the bytecode (i.e., `J_Guard`) and the native code (i.e., `N_Guard`). Each goal must be controlled at least by a guard (`Guard-Goal`) and each guard must be likewise controlled by at least two other guards (`Guard-Guard`) to improve redundancy and reliability to attacks. According to previous rules, AppIS generates a set of Guard-Goal and Guard-Guard constraints (Step 2.4), which are taken into consideration to build a set of *guarding nets* among goals and guards, that satisfies all constraints. Moreover, such set is used to construct a scheduling engine that at runtime chooses which guarding net to activate, according to the triggered goal and the execution status. Finally, AppIS builds and delivers the protected *apk*, containing goals, guarding nets and the scheduling engine.

Runtime Behavior: At runtime the scheduler generates a guarding net (Step 6.1). In this way, a different guarding net is generated at each execution, in order to make the app resilient against cumulative attacks (i.e., multiple execution of the app in a sandboxed environment to detect guards). During the execution (Step 6.2), whenever a guard is triggered (Step 6.3), the corresponding guards check the integrity of the goal (i.e., `J_Guard`) or the guard (i.e., `N_Guard`) (Step 6.4) by comparing the checksum of the goal calculated during the static construction (i.e., Steps 2.3 to 2.5) with the checksum obtained at runtime on the same goal (Step 6.4). If the two checksums do not match, each guard detects a repackaging and the app is directly terminated.

Evaluation: AppIS improves the previous proposals by adding networks of guards and a non-deterministic behavior at runtime, that complicates the attacker analysis activity. AppIS has been evaluated on a very reduced set of only 8 apps,

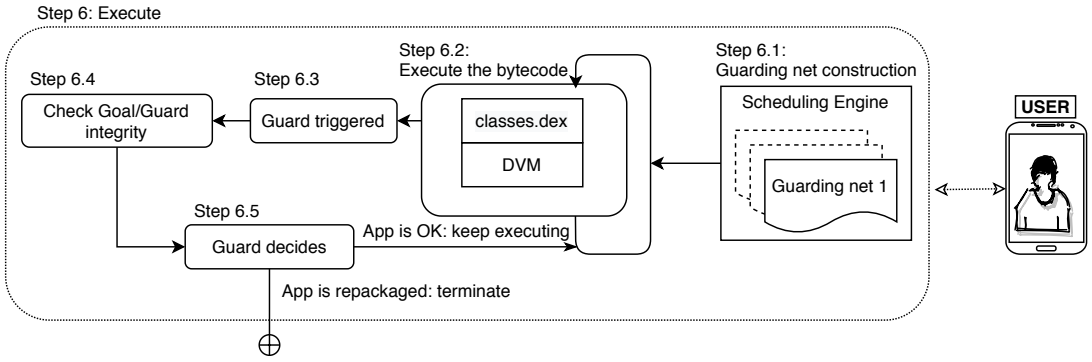


Figure 3.6: AppIS: runtime behavior.

collected from different sources, on almost outdated Android versions (spanning from 4.4 to 6). Experimental results indicate that no app fails, and overheads are reasonable w.r.t. the previous proposals; in fact, the space overhead is up to 2% in the worst case, while time overhead spans from 2% to 140%. However, we argue that the reduced number of apps is statistically insignificant. The reliability of AppIS has been tested against two threats, namely the possibility for an attacker to i) obtain the collection of the guarding nets, and ii) to carry out cumulative attacks. According to the authors, in both cases the AppIS approach revealed to be reasonably robust.

3.4 Self-Defending Code through Information Asymmetry

In 2018, Kai Chen et al. in [35] proposed a self-defending code (SDC) approach, leveraging the information asymmetry between the developer and the attacker on the app code, i.e., it is reasonable to assume that the attacker has far more less information than the developer on the app code, on average. Similarly to the approach discussed in Section 3.1, the idea is to encrypt pieces of source code

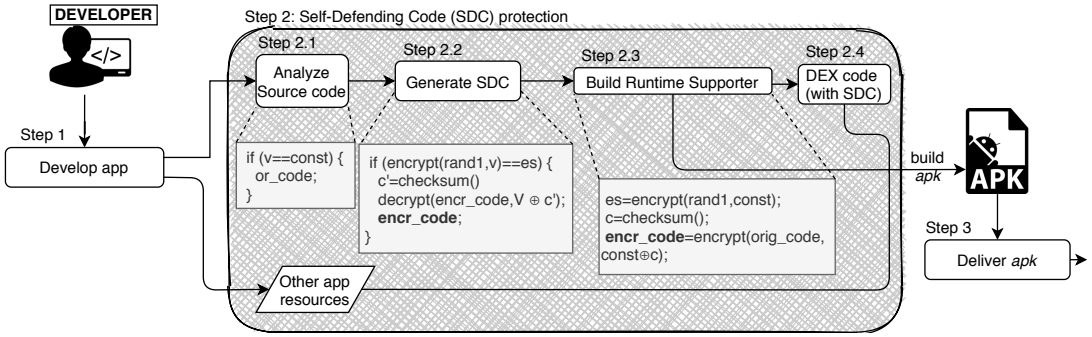


Figure 3.7: The SDC protection scheme.

and decrypt (and properly execute) them at runtime only in case the app has not been repackaged. Unlike the previous work, each piece of code is encrypted with a different key. The approach provides two schemes, where the first one requires to customize the Android OS, and the second one can work on unmodified OSes. We will focus on the latter one, as we argue that the first scheme cannot scale in the wild, due to the customization needs.

Protection scheme: The scheme (see Figure 3.7) begins by analyzing the source code (Step 2.1) and selecting some qualified conditions (i.e., branches containing an equality check, where one of the operands is a constant value: i.e., `v==const`) in the code that need to be protected. A qualified condition is then substituted by a self-defending code (SDC) (Step 2.2) that hosts the encrypted version of the original code contained in the body of the qualified condition. Once all the pieces of code have been encrypted, a component named *runtime supporter* is built. Such component contains all the encryption routines which allow understanding how the modified code has been generated (Step 2.3); such routines are useful for reconstructing the original code at runtime. It is worth pointing out that the encryption scheme is XOR-based and the key is related to the checksum of the original piece of code. Therefore, the original code can be decrypted correctly if and only if the app has not been modified. More in detail,

w.r.t. to the sample code at Step 2.1 in Figure 3.7, the information asymmetry between the developer and the attacker is the knowledge of the value `const`, removed from the source code at compiled time in Step 2.2, and the `encr_code`. Furthermore, `encr_code` is redundantly protected by a set of other - randomly selected - SDC segments, some of which implement part of their functions as native code. Finally, the protected apk, composed by the DEX code extended with the SDC, the non-code resources and the runtime supporter, is built and delivered.

Runtime Behavior: At runtime, whenever an SDC is executed, the encrypted code is decrypted by leveraging the app checksum xored with the `v` value as shown in Step 2.2. If one of these differs from the original values, the decryption fails and the original code is not retrieved, thereby leading the app to crash.

Evaluation: Differently from previous proposals, SDC has been validated on a significant set of apps (i.e., 20,000) taken from Anzhi [36] and belonging to 15 categories. For each app, a set of more than 100 candidate branches has been selected and substituted with SDC segments. The testing phase involved both real users and automatic testing. In the first case, ten SDC segments where triggered in an hour, while, in the latter case, five segments were triggered within 24 hours. The time overhead is below 4% for each app tested, while the space overhead is negligible.

3.5 BombDroid: Decentralized Repackaging Detection through Logic Bombs

In 2018, Zeng et al. in [37] introduced the concept of *logic bombs* which are hidden (i.e., cryptographically obfuscated) pieces of code that execute once proper triggers (i.e., logical conditions) are activated. The authors implemented the approach in a tool called BombDroid. Like SDC (Section 3.4), logic bombs are injected in qualified conditions.

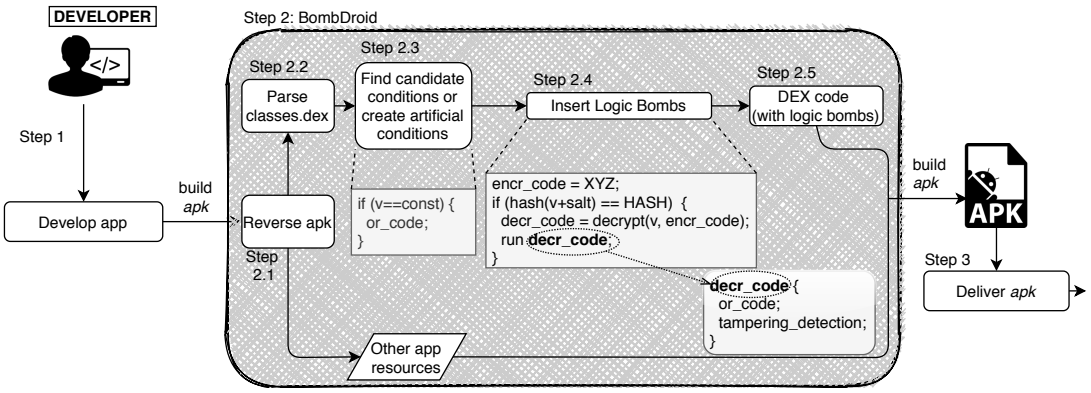


Figure 3.8: The BombDroid protection scheme.

Protection scheme: The protection scheme of BombDroid (see Figure 3.8) is very similar to the SDC one, and the idea is to selectively find some candidate condition in the app methods to inject logic bombs. The first difference is that BombDroid works directly on the *apk* at the bytecode level, and not on the source code. The BombDroid protection scheme begins by reversing the *apk* (Step 2.1), and parsing the `classes.dex` file (Step 2.2) through static analysis techniques to identify suitable methods (Step 2.3). In each selected methods, BombDroid identifies at least a qualified condition containing an equality between a variable and a constant value (`v==const`, see Figure 3.8). The original code - contained in the corresponding block - is extended with some checks aimed at detecting

tampering, and then, encrypted. The encrypted code will be decrypted at runtime according to the hash value of the variable plus a salt, selected by BombDroid. The decryption key is the value of the constant `v`. The use of a salt, instead of the checksum of the original method, is the main difference with SDC. The robustness of the scheme is likewise given by the robustness of the hash function, that do not allow guessing the value of `v`. Moreover, it is worth pointing out that the selected methods can have no qualified condition in principle: in this case, BombDroid can inject artificial qualified conditions to protect. Finally, BombDroid allows nesting logic bombs to improve stealthiness: for instance, with reference to the logic bomb in Step 2.4 of Figure 3.8, another encrypted logic bomb can be inserted in the decrypted code between the original code and the tampering detection function.

Runtime behavior: At runtime, the logic bomb is executed, and the code is properly decrypted only if the value of the variable in the qualified condition is equal to `const` [1] and the logic bomb activates. Then, a tampering detection function is executed. BombDroid applies three anti-tampering techniques, i.e., at app level, at file level and at code snippet level. At compile time, BombDroid stores the original public key in the *apk*, as well as the values of some digests concerning resources and the `classes.dex` file, and some code snippets. At runtime, tampering functions calculate the digests of such resources and compare them with the stored one: tampering is detected whether one of such value differs from the stored one. In case a tampering is detected, the user could be warned or the app could cause negative user experience.

Evaluation: Authors assessed the reliability and the performance of BombDroid on a set of apps (i.e., 963) taken from F-Droid. They exploited Dynodroid

[1] This condition triggers the execution of `or_code` in the original code of the candidate method.

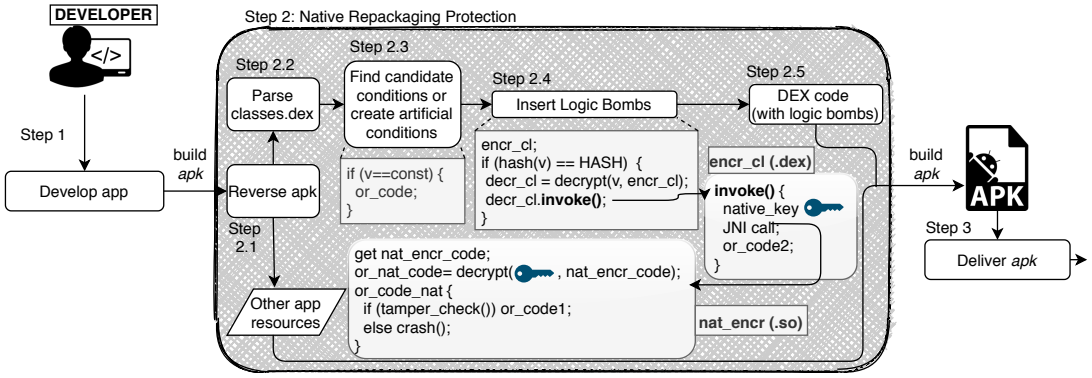


Figure 3.9: The Native Repackaging Protection scheme.

[38] to execute each app for an hour, repeating the experiment 50 times per app. The execution time overhead is almost negligible (i.e., ¡2.7%). The space overhead ranges from 8% to 13%. On average, the first logic bomb is triggered between 75 and 164 seconds after the app starts executing, and on average, only the 9.3% of logic bombs are triggered.

3.6 Native Repackaging Protection

In 2019, Simon Tanner et al. in [39] proposed an evolution of previous techniques. A prototype of this work is available on GitHub [40]. Like BombDroid, this work is based on cryptographically obfuscated logic bombs, inserted in candidate methods with a qualified condition (i.e., `v==const`). However, the main difference is that integrity checks and a part of the original app code of the block are executed in native code. We refer this scheme as *Native Repackaging Protection*[1] (NRP).

Protection scheme: The NRP protection scheme, which mostly resembles the BombDroid and SDC ones, is depicted in Figure 3.9. Also in this case, NRP reverses the *apk* (Step 2.1), parses the `classes.dex` file (Step 2.2), and finds (or

[1] Authors do no provide a name for this scheme.

builds artificial) blocks with a qualified condition. Then, it modifies each selected block as follows: i) given the standard condition `v==const`, NRP calculates the hash value of the constant (i.e., `HASH=H(const)`) using SHA-1 as the cryptographic hash function. Then, ii) the instructions of the block code (i.e., `or_code`) are inserted in a new Java class which is encrypted using AES-128 in CTR mode, using the *const* value as key. NRP applies several workarounds to preserve the semantic equivalence between the original code and the code contained in the encrypted Java class. Such class (i.e., `encr_cl`) contains a single method `invoke()` which splits the original block code into two parts, namely `nat_encr` and `or_code2`. The first part refers to a native library which is likewise XOR-encrypted with a random symmetric key (i.e., *native_key*). Such key is stored in the same encrypted Java class. To decrypt and execute the function `or_nat_code()`, stored in the `nat_encr` library, a JNI call is required. `nat_encr` contains an anti-tampering check that executes the first part of the original block code only if no tampering is detected. In current implementation, the integrity check verifies the signature of the first 100 bytes of the `classes.dex` file, whose path is hard-coded in the native library.

Runtime behavior: During execution, the triggering of a logic bomb, depicted in Step 2.4 in Figure 3.9, leads to the following execution: i) once the condition on the hash value is verified (i.e., `hash(v)==HASH`), the encrypted Java class (`encr_cl`) is decrypted according to the constant value, and then the `invoke()` method is executed through Java reflection. The first statement (`nat_encr`) is executed, and leads to the decryption of the native library (`nat_encr`) and the execution of the `or_code_nat` block. Then, the signature of the first 100 bytes of the `classes.dex` file are checked. If the verification succeeds (i.e., `tamper_check()==true`), then the first part of the original block code is executed (`or_code1`),

otherwise the app crashes. After the proper execution of `or_code1`, the execution flow is returned to the `invoke()` method which executes the second part of the original block code (i.e., `or_code2`).

Evaluation: NRP has been tested against 100 apps downloaded from the Google Play Store and F-Droid. Tests were performed on an LG Nexus 5X device, running Android 8.1.0. In their implementation, authors leverage the `dalvik.system.InMemoryDexClassLoader` to load the new Java class of each encrypted block. This functionality is available starting from Android 8.0, only. The empirical assessment showed that 47% of apps could be transformed without runtime exceptions. Each app has been tested through Monkey [32] to generate random user's input: the time overhead is from 1.42% up to 10.42% (the space overhead is negligible), while on average, from 11 to 35 logic bombs have been activated.

Chapter 4

Attacking Anti-Repackaging

Summary

In this chapter, we discuss the *weapon rack* of repackaging, i.e., the main set of techniques that an attacker can leverage to analyze a protected *apk* in order to detect and dismantle anti-repackaging protections (Steps 8 and 9 in Figure 2.1). They can be divided in two main categories, namely *static* and *dynamic*. The first category refers to the set of techniques that analyze the *apk* without executing it (static analysis), while the latter category is composed by techniques that apply to an *apk* while executing on an actual or emulated Android device (dynamic analysis).

On one hand, static analysis aims at systematically reverse engineering the *apk* (e.g., White Box Reverse Engineering [41]) in order to reconstruct and analyze the original source code. There exist a lot of tools that support this type of analysis and help an attacker to decode, rebuild and inspect both the app code and the resources, such as apktool [14] and dex2jar [42]. On the other hand, dynamic analysis aims at inferring the app behavior during execution, to retrieve, e.g., actual values of variables, and monitor the execution of the code.

Static and dynamic analysis are complementary: the first aims at building a comprehensive model of the app behavior by systematically exploring all the code and the resources of the app; the latter focuses on inferring a model of the app behavior by repeatedly executing the app. In principle, the static model is a superset of all possible behaviors of the app, meaning that at runtime the app will exhibit only a subset of the behavior of the static model. The dynamic model is partial by definition (as it is impossible to stimulate all possible executions of the app at runtime) but contains only behaviors that the app actually exhibits. As a consequence, static analysis suffers from *false positives* (e.g., a statically detected behavior could not be executed at runtime), while dynamic analysis suffers from *false negatives* (a behavior which has not been recognized during the dynamic analysis phase could be assumed as non-existent).

Concerning anti-repackaging, static analysis aims to detect the pieces of code where logic bombs could potentially hide (e.g., blocks with hash values in the qualified condition), while dynamic analysis allows to check the actual triggering and execution of the bombs at runtime. Furthermore, the repeated execution of the app on several setups allows the attacker to carry out *cumulative attacks* [34][1]. Within the "try and error" cycle, the attacker builds up an analysis workflow of the *apk* which fruitfully combines static and dynamic analysis techniques.

In 4.2 we will then discuss how the attacking techniques previously shown can be leveraged to disable the anti-repackaging techniques presented in Section 4.1, and to build a fully working repackaged app.

Unfortunately, the vast majority of the older anti-repackaging techniques (i.e., described in Sections 3.1, 3.2, 3.3, 3.4 and 3.5) have no available implementation to date. We tried to contact their authors via institutional emails, but we did not receive any positive answer (i.e., neither the source code nor a sample of a

[1]See the AppIS approach in Section 3.3

protected app). For this reason, we will just provide some motivated guidelines on how such techniques can be circumvented.

However, the source code of the native extension of BombDroid [39], described in Section 3.6, is available as source code on GitHub [40]. For this latter and more recent proposal, we describe a full-fledged attack able to circumvent all the anti-repackaging checks.

It is also worth noticing that none of the anti-repackaging techniques carries out any anti-debugging or emulation check. This basically simplifies the execution of cumulative attacks because the attacker is able to perform tests without worrying about emulator or debugging detection.

4.1 Attack Model

4.1.1 Text Search and Pattern Matching

This technique statically analyzes the decompiled *apk* to detect keywords or expressions that may hide logic bombs. The search and detect phase relies on heuristics and regular expressions defined by the attacker herself, or on pre-defined patterns defined in static analysis tools that support this technique. For instance, this technique may allow recognizing explicit anti-tampering function calls, as well as qualified conditions containing hash values. As any other static analysis approach, all detected anti-tampering checks must be evaluated at runtime to filter true positives (i.e., the checks which actually hide some logic bombs).

The efficacy of this technique is rather limited, as it can be easily circumvented by anti-repackaging techniques. For example, SSN and BombDroid [33; 37] are resilient to this threat as they rely on Java reflection or obfuscation [43]; the first allows hiding function calls inside the code, while the latter avoids text search as obfuscation modifies the syntax of the code (e.g., method renaming).

4.1.2 Fuzzing

Fuzzing is an automated testing technique that provides pseudo-random input to a running program. There exist three different types of approaches, namely:

- **Blackbox fuzzing**: it generates input data without having any knowledge of the app code and structure [38];
- **Whitebox fuzzing**: it is also known as *symbolic execution* [44; 45] and it applies to the app code, which is supposed to be accessible by the attacker. The main goal of whitebox fuzzing is to provide input data which allows exploring as many execution paths as possible, in order to detect which data could match specific condition (e.g., the trigger of a logic bomb). Moreover, whitebox fuzzing could allow revealing the destination of a suspicious call executed through Java reflection. From a technical standpoint, the main difference between blackbox and whitebox fuzzing is that the latter adopts a (symbolic) input which evolves according to the program analysis and to reach certain critical program location; the first one, instead, does not rely on the program structure.
- **Graybox fuzzing**: this approach mixes blackbox and whitebox fuzzing, i.e., it generates symbolic input for the sole subpart of the program which is known to the attacker, while the unknown part of the app is stimulated according to blackbox fuzzing data generation.

Fuzzing is suitable for assessing an *apk* protected with SSN or Bombdroid, as it allows to trigger (and detect) the bombs (repackaging detection nodes) or to perform analysis of hash values. However, these techniques do not grant a complete path exploration and a full input data coverage. Therefore, they cannot find and trigger all bombs [43].

4.1.3 Debugging and Code Emulation

The most common way to deeply analyze the behavior of an app is to use a debugger. In the general case, a debugger allows a developer to dynamically test a software in order to find and resolve bugs. Concerning the attack to anti-repackaging, a debugger allows bypassing all static countermeasures like obfuscation, and understand how the app behaves at runtime. More in detail, an attacker can intercept critical calls, find out repackaging detection and understand the type of anti-tampering checks. However, debugging strongly relies on manual intervention, which has the drawback to scale badly, i.e., debugging an app to try tracing back all bombs has been proven to be unfeasbile [37; 43].

In addition, the debuggable flag inside the app manifest file has to be set to true, in order to attach a debugger to an Android app. Developers that apply anti-repackaging techniques trivially set this flag to false. Therefore, an attacker should first modify the app manifest [46] and repackage it before debugging. This first repackaging step can itself trigger proper logic bombs in the app and make the execution of the app fail.

An alternative to debugging is *code emulation* in [47; 48]: the idea is that the target app is executed on some virtual machine, where the hardware, the Android OS and the Dalvik VM layers are replicated. During app execution, the virtual machine performs call tracing [49]. However, both approaches follow specific execution paths that depends on input data and could not reach the candidate methods in which logic bombs are hidden.

4.1.4 Process Exploration

Process Exploration refers to a set of techniques allowing the attacker to deeply inspect the app process memory at runtime [46]. There exists several tool to sup-

port memory exploration, most of which are Frida-based[1], and rely on *dynamic code instrumentation*. Currently, the most widespread tools are:

- **Fridump**: an open source memory dumping tool for both Android and iOS [50]. From a memory dump, it is possible to access to memory addresses where, for example, decryption key of anti-repackaging tools are stored at runtime;
- **Objection**: a runtime mobile exploration toolkit for both Android and iOS, that does not require a rooted device. Two interesting features of Objection are the ability to i) circumvent SSL pinning, i.e., a technique allowing apps to authenticate the server and avoid man-in-the-middle-attacks), and ii) dump and patch memory locations [51];
- **R2frida**: a tool that merges the reverse engineering capabilities of *radare2* [52] (a reverse engineering framework) with Frida (a dynamic instrumentation toolkit). In this way, an attacker can carry out runtime reverse engineering and memory related tasks, such as memory inspection [53].

The adoption of such tools allows an attacker to analyze the runtime behavior of an app without debugging it (i.e., without the need to force the set of the debuggable flag to true and repackage.).

4.1.5 vtable Hijacking

In Java, a virtual table (vtable) dynamically maps function calls, i.e., a Java object points to some records of the vtable which contains the link to the actual method implementation. In Java, all methods are virtual by default, unless they are declared as *final* or *static*. An attacker can modify the values in the

[1]We will discuss Frida in Section 4.1.6

vtable in order to invoke an arbitrary method and change the behavior of the original method. The limitation of this approach is that it works properly only on rooted devices, as the attacker needs root privileges to change the entries of the vtable [54].

An example of vtable hijacking is proposed in [33]: here, the vtable mapping is modified to redirect all invocation to the *getPublicKey* method to a spoofed function which returns the public key of the original developer, instead of the public key of the attacker that repackaged the app.

4.1.6 Code Manipulation

There exist three main techniques to affect the app code in order to remove anti-repackaging checks or to add malicious code, namely *code deletion*, *binary patching* and *code instrumentation and injection*. The first two techniques are static, while the latter is dynamic.

Code deletion: This technique aims at removing proper part of the app source code (if available), or smali code in order to deactivate specific controls. It can be partially circumvented: for instance, the anti-repackaging checks may be hosted in methods that cannot be straightforwardly removed, i.e., many other parts of the app depend on them. Alternatively, fake controls resembling logic bombs can be added in the code to fool the attacker which exploits code deletion [37]. In case the attacker remove them from the code, the app will not work properly.

Binary patching: It refers to the process of modifying a compiled executable to change its behavior. Differently from code deletion, binary patching does not need to retrieve or reconstruct the app source code, and it is, for instance, independent from code obfuscation. It is worth pointing out that it is far easier

and more reliable to apply binary patching to Java than native code (e.g., C, C++, ...). Therefore, this technique is promising for an attacker, as current apps still contain most of the business logic in the Java/Kotlin part of the app (see Section 2) rather than in the native part.

To counteract binary patching, the anti-repackaging checks need to be protected, i.e., through encryption applied at Java (see Sections 3.1, 3.4, 3.5) or native level (Section 3.6). As an alternative, checks can be hidden by exploiting Java reflection [33], however, once the attacker understands the protection pattern, he could patch the executable and remove checks with minimum effort, as discussed in [37; 43].

Code Instrumentation and Injection: Such techniques allow to dynamically modify the code of a running process. In this way, the attacker is able to analyze functions or system calls, as well as their parameters and return values, and modify the behavior of the original functions.

There exist a lot of tools that support code instrumentation and injection, as well as the corresponding runtime analysis. Nonetheless, Xposed [55] and Frida [16] are currently the *de-facto* standards. Xposed is a framework that applies add-ons (called *modules*) directly to the Android OS ROM and requires root privileges. Such modules may allow, e.g., customizing the script that spawns any new process from the Zygote one[1], in order to add a jar file that allows hooking.

Frida is a dynamic code instrumentation tool that allows an attacker to hook functions by injecting a JavaScript engine into the instrumented process; the engine then allows injecting executable code by directly modifying the process memory. Frida can work on both rooted and unrooted devices in three different modes:

[1]The Zygote process is responsible for spawning new process to host launching apps.

- *Injected mode*: this is the most common deployment in which the user installs and runs the frida-server (i.e., a daemon) into a rooted device that exposes the frida-core module over TCP (usually on port 27042).

- *Embedded mode*: the user repackages the app with the frida-gadget (i.e., a shared library). This library allows instrumenting the app on an unrooted device.

- *Preloaded mode*: the frida-gadget shared library is injected inside the operating system layer instead of the application layer (as in the embedded mode). It works on rooted devices only.

Code instrumentation and injection allow stealing the actual values of program variables from a running application. For example, the attacker could hook the function that decrypts some portion of code. In this way, he is able to recover the key for that specific chunk and the plain text. Then, the attacker can replace the encrypted code of an anti-repackaging check with a crafted one, by exploiting the binary patching technique previously discussed [56].

From the anti-repackaging standpoint, it is a hard task to safely hide checks that are resilient to code instrumentation and injection. As a matter of fact, system call hooking allows in principle to detect and attack any anti-repackaging check even if it is based on some kernel-provided functionality. In fact, Frida allows hooking tampering detection functions, and makes them return the attacker's desired value. In addition, some "anti-frida" controls (e.g. simply check if the tcp port 27042 is open) relies on system call and can be circumvented by Frida itself.

An attempt to limit function hooking is proposed in [39], where some Java instructions are translated into native code to elude Java code instrumentation and avoid an attacker to recover the original statements.

4.1.7 Network-based attacks

Previous attacking techniques focus on the device, the OS and the app. However, it is also possible to attack anti-repackaging at network level. Such attacks can be passive, i.e., the attacker does not affect the network traffic (e.g., *sniffing attack*) or active, i.e., the attacker willingly affect and spoof the traffic, (e.g., *replay attack*).

Sniffing attack. It refers to the interception and the analysis of data exchanged by two parties on a communication channel (e.g., an app interacting with a remote server) [57; 58]. This attack is relevant as some anti-repackaging checks are executed at server side. If an attacker understands the communication pattern and which data are sent between parties, he could be able to detect and circumvent these controls [59]. For instance, an attacker is able to read the client-server communication if it is over *http*.

Replay Attack. The replay attack (or Man in the Middle attack - MitM) extends the sniffing attack by actively interacting with the two parties. The attack dynamically affects the data exchanged during the communication, by relying on a MitM proxy [60]. Concerning anti-repackaging, if the signature check of an *apk* is carried out at server side, then the client has to send the checksum of its certificate and an attacker could change the communication data in order to trick the server, or spoof a fictitious response from the server. Albeit all anti-repackaging techniques discussed in Section 3 leverage local checks only, there exists some proprietary solutions which provides remote anti-repackaging detection, like Google safetyNet [61]. The safetyNet API is provided by Google to Android developers, and can be included in any app. Such API provides functionalities against several security threats and specific checks against repackaging, which are applied

both locally e.g., emulator detection, and remotely, e.g., signature detection on the server side. In this case, the attacker can try modifying the network traffic generated by the safetyNet API calls, to bypass server-side checks or spoof a fake answers from the server. Furthermore, the attacker can remove the invocation to safetyNet API functions by directly patching the smali code (or native one) by leveraging code deletion (Section 4.1.6).

It is worth pointing out that MitM attacks can be carried out also inside the *apk* itself: in this scenario, the attacker could modify at runtime some data using some of the code/data manipulation techniques previously discussed, with the aim to bypass specific anti-repackaging checks. The main idea is that an attacker could replace some resource file in order to bypass specific controls which rely on the app data (e.g., modify the path from the repackaged resource into the original one) [35; 43].

4.2 Disabling Anti-Repackaging

4.2.1 Self-Protection through *dex* Encryption

The encryption method used to cipher the `classes.dex` file is a XOR encryption. This method could be easily extended to adopt more secure ciphers, like, e.g., AES; nonetheless, this would increase the runtime overhead.

Independently from the cipher, the main drawback of the proposal is that the decryption key of the ciphered bytecode (i.e., the `classes.dex` file) is hardcoded in an obfuscated (via LLVM/OLLVM) native activity. However, there is no need to statically de-obfuscate the native code to retrieve the key, as three dynamic analysis attacking techniques can be leveraged:

- symbolic execution could reveal the decryption key and how it is obtained. This technique allows tracking several execution flows showing the value of

the variables, including the decryption key;

- the debugger allows to detect the actual value of the key at runtime. To debug the app, the only modification is to change the value of the debuggable flag into the *AndroidManifest.xml* file. This tampering is not detected by the XOR-based checksum at runtime;

- code instrumentation could allow retrieving the decryption key which is passed as parameter to the decryption function. Function calls could be intercepted with different tools (e.g., see the Frida script in Figure 4.1) in order to report back the the values of its arguments.

```
1   function inspectModule(module) {
2     var m = Process.getModuleByName(module);
3
4     Interceptor.attach(m.getExportByName("decrypt"), {}
5       onEnter: function(args) {
6         // Dump arguments
7         send("Argument 1 of decrypt function" , args[0]);
8         send("Argument 2 of decrypt function" , args[1]);
9         [...]
10      }
11    });
12  }
```

Figure 4.1: Frida script to dump arguments provided as input to the native method *decrypt*.

The access to the decryption key allows the attacker to decrypt any slice of ciphered code. Furthermore, cumulative attacks can be performed to retrieve all the decrypted code using runtime memory dumping and process exploration. In this scenario, the attacker can dump each decrypted bytecode at runtime, thereby recovering the original code and she can leverage code manipulation (e.g., code deletion or binary patching) to add malicious code or remove anti-tampering controls.

Once the code is recovered and tampered, one of the following approaches can be leveraged to create a working tampered *apk* based on code deletion and binary patching:

- override the body with the encrypted tampered code. This approach can be used only if a symmetric-key algorithm is used to encrypt and decrypt the code. In this way, once the attacker recovered the encryption key, she can perform both encryption and decryption.
- replace directly the body with the tampered code. In this scenario, the final app does not contains encrypted code.

4.2.2 Stochastic Stealthy Networks

Several works [37; 39; 43] have already pointed out some weaknesses of SSN, as well as some attacks that can be used to circumvent this protection. There are two ways to circumvent SSN, namely i) deactivating the integrity check or ii) detect and remove all the detection nodes from the code.

Concerning the first strategy, recall that each detection node implements the same integrity check on any candidate method, based on the developer's public key. Therefore, it is sufficient to analyze a single detection node to define a de-activation strategy. To this aim, code injection or vtable hijacking allow replacing the target method (*getPublicKey* and *generateCertificate*) with a custom implementation, which returns the original developer's public key, instead of the attacker's one.

Regarding the latter strategy, blackbox fuzzing can be used to generate a massive number of pseudo-random input in order to trigger all repackaging detection nodes (which are invoked stochastically) to detect them. In addition, whitebox fuzzing (or symbolic execution) can reveal the destination of the reflection calls.

Once all detection nodes are detected, either code deletion (i.e., to remove the repackaging detection nodes) or binary patching (i.e., to make the repackaging detection node unreachable) can be leveraged.

4.2.3 AppIS: Protect Android Apps Against Runtime Repackaging Attacks

The simplest way to repack an AppIS-protected app is to modify only parts (methods, resources or data), which are not protected by guards. It is possible to recognize whether an app snippet is protected through static analysis, as no code obfuscation is applied by AppIS. Static analysis also allows obtaining the set of guarding nets, and delete them from the code. This is made possible by the fact that each guard has a specific design pattern which the attacker can detect through bytecode analysis. In future works, the authors put forward the possibility to encrypt the set of guarding nets and decrypt it at runtime. Nonetheless, the encrypted guarding nets would be likewise vulnerable to deletion attack, as it is sufficient for the attacker to remove the parts of code where the decryption is carried out.

Beyond removing guarding nets, it is also possible to bypass the integrity detection on the goals, by exploiting cumulative attacks to collect a large number of the original hash values. Then, such values can be hardcoded in the corresponding integrity detection methods. In this way, the attacker is able to bypass the anti-tampering controls without deleting the guarding nodes.

4.2.4 Self-Defending Code through Information Asymmetry

Recall that this approach is an improvement of the previous one: it moves the integrity checks to the native code and uses cryptographic obfuscation to resist static analysis. Nonetheless, it remains vulnerable to several dynamic analysis techniques: first, blackbox fuzzing allows triggering a large number of SDC segments. For each triggered segment, a code instrumentation tool like Frida can be leveraged to hook the `encrypt` function in the `if` statement in order to recover the correct value of the variable `v`. Once this data is obtained, it is possible to retrieve the decryption key of the encrypted code, in both modes supported by this anti-repackaging technique. Then, code deletion can be leveraged to substitute an SDC segment with the original code, thereby deactivating the protection. Previous attacks succeed only if all SDC sufficient for repackaging succesfully are discovered. Therefore, the robustness of this anti-repackaging technique is directly dependent on the number and the distribution of SDC statements in the app.

Finally, if the attacker is able to locate the checksum method in the native code (the anti-tampering control), she would override the body of such method through binary patching, in order to force it returning the excepted value. In this way, we would be able to tamper an application and bypass all checks with low effort.

4.2.5 BombDroid: Decentralized Repackaging Detection through Logic Bombs

In BombDroid, any logic bomb follows the same, well-defined, specific pattern (i.e., `hash(v)==const`). Text search and pattern matching can be used to stati-

cally analyze the app source according to this pattern, in order to locate potential pieces of codes hiding logic bombs. However, code obfuscation could complicate this pattern-based search.

Once a potential logic bomb is detected, the attacker can try to retrieve the decryption key (which is the constant value of the qualified condition) through dynamic analysis by leveraging:

- **Brute force**: The attacker can try all possible values for the integer value `v`. Potentially, this key space is huge (i.e., 2^{32}, as Java stores integer values in 4 bytes). Nonetheless, the actual distribution of valid constant values is smaller [40]. Furthermore, there is a small number of unique constant values (i.e., in the range [-1024, 1024], according to the authors).

- **Extraction of information from open source code**: If an app includes external open source libraries, and the static analysis recognize some potential logic bombs on such libraries, then an analysis of the original source code can allow guessing the right constant value of `v`.

- **Code instrumentation**: Proper code instrumentation applied on the potential logic bombs can allow intercepting the call to the decryption function (e.g., AES), thereby recovering the decryption key or the plain code (once the logic bomb has been triggered).

Another way to retrieve the value of `v` is by means of cumulative attacks. In fact, as authors showed that at most 9.3% of bombs are triggered using both state of the art Android fuzzing tools and human analysis, the detection and de-activation of such few logic bombs can suffice to obtain a repackaged app that works properly in the most of the cases. Put another way, this fact indicates that albeit BombDroid can add a lot of logic bombs, only few of them actually trigger.

Therefore, it is unnecessary for an attacker to delete each logic bomb, but she needs to remove just the mostly triggered ones on average.

Once a constant value is retrieved from a single-trigger bomb, the logic bomb could be deleted (i.e., deletion attack) and replaced directly with the (original) decrypted code. The attacker can also add some extra code to the original one. It is worth noticing that the previous attacking procedure applies on nested bombs if all constant values (i.e., contained in all nested qualified conditions) are discovered.

Another possibility is to bypass the anti-tampering checks contained in the encrypted code of the bomb (i.e., the `tampering_detection` function in Figure 3.8):

- leverage code injection to replace the tampering detection function with an attacker-defined one;
- try a replay attack to modify the MANIFEST.MF file used in the code digest comparison checks. In this scenario, the anti-tampering controls compare the hardcoded values with the one in the original MANIFEST.MF file.

Recall that the BombDroid prototype relies on a tampering detection function based on a public-key comparison. Such function can be dynamically rewritten by instrumenting the Dalvik bytecode in order to return the expected value (i.e., the public-key of the original developer).

4.2.6 Native Repackaging Protection

The Native Repackaging Protection is the most recent and advanced repackaging avoidance technique to date, as it inherits and extends methodologies and techniques from the other proposals. For previous anti-repackaging techniques, we have pinpointed some high-level attack patterns by leveraging the attacking

techniques described in Section 4.1, since no source code, nor any protected app have been made available. However, as the source code of NRP is publicly available, we were able to analyze it in detail and carry out an actual and full-fledged attack to this technique. We will describe the attack in details by relying on a real app, i.e., Antimine [7], an open source, minesweeper-like game app available on F-Droid (see Figure 4.2).

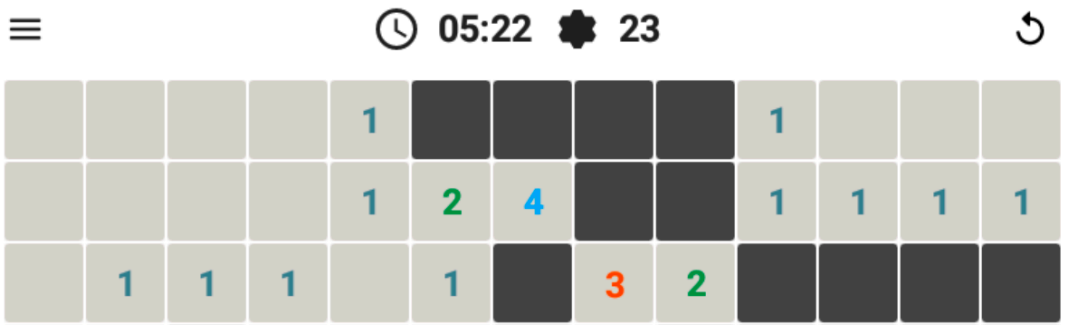


Figure 4.2: A snapshot of the Antimine game.

More specifically, we will i) apply NRP to Antimine, showing how a candidate method is protected, ii) prove how such protection can be circumvented and iii) build a full-working repackaged version of the protected Antimine app.

As a preliminary remark, we found an implementation bug by analyzing the NRP source code: the tool statically pre-computes the signature of the `classes.dex` file through a Java function, while at runtime the protected app calculates this value by means of a native function (C/C++ code). The `char` type in standard C can either be signed [-128, +127] or unsigned [0, 255]. We found that the native function uses a char (line 3 on the left side of Figure 4.3) while the corresponding Java function adopts an unsigned char (line 4 on the right side of Figure 4.3). As a consequence, a char value that has a bit representation with a `1*******` pattern (where `*` stands for 0 or 1) is interpreted as an int value between [-128,-1] by C and between [127,255] by Java. Therefore, if some chars of the `classes.de`

```
.
1  // XOR in native code
2  int32_t hash_bytes(const char *arr,
       size_t count) {
3    char hash[4];// <-- Should be u_char
4    for(size_t read = 0; read < count;
         read++) {
5      hash[read % 4] ^= arr[read];
6    }
7    return hash[0] + (hash[1] << 8) + (
         hash[2] << 16) + (hash[3] << 24)
         ;
8  }
```

```
1  // XOR in Java code
2  private static int xorFile(Path p, int
       count, int offset) throws
     IOException {
3    byte[] content = Files.readAllBytes(
       p);
4    byte[] res = new byte[4];
5    for(int i = offset; i < (offset +
       count); i++ ) {
6      res[(i - offset) % 4] ^= content[i
         ];
7    }
8    return (0xff & res[0]) + ((0xff &
       res[1]) << 8) +
9      ((0xff & res[2]) << 16) + ((0xff &
         res[3]) << 24);
10 }
```

Figure 4.3: On the left side: the C code used to calculate the hash sum at runtime. On the right side, the Java code used to precompute the hash sum, at compile-time.

`x` file have the `1*******` format, the integrity check always fails - independently from the app being tampered or not.

Moreover, the translation of the Java bytecode into native code could introduce some new exception. For instance, any single instruction in the Java bytecode influence, or is influenced, by other instructions (for example the comparison between two variables). It is mandatory to manage this mutual relationship, and the program must behave soundly even if some instructions are executed under the native environment. We argue that this is a non-trivial and error-prone task, as no automatic tool currently exists to translate Java bytecode to native on Android. Furthermore, NRP suffers from the same limitations of BombDroid, related to the use of hash functions to hide the decryption key. If a logic bomb is inserted in an `if` statement with a small hash space, it could be rather quick to recover the decryption key by brute force. As the authors discuss, NRP suffers from this limitation, as most of the constant values in an *apk* are in the interval [-1024, 1024]. Finally, and differently from BombDroid, the hash function in NRP is not salted. As a consequence, whenever a decryption key is recovered,

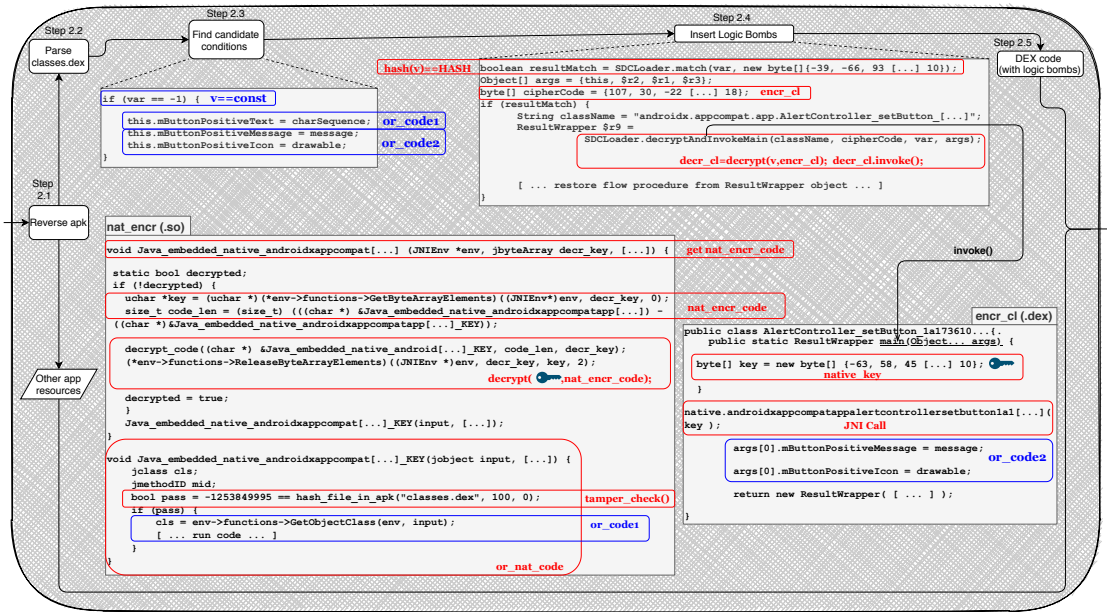


Figure 4.4: NRP applied to the `setButton()` method of Antimine.

it is possible to statically detect all logic bombs that have the same key by only comparing the hash values. In fact, if two candidate conditions have the same constant value (e.g., `if(a == 1) ...` and `if (b == 1)...`, the hardcoded values in the trigger condition are likewise the same (i.e., `if(hash(a) == 'xyz')` and `if(hash(b) == 'xyz')`, since no salt is used.

4.2.6.1 Protecting Antimine with NRP

We applied NRP to Antimine: the process lead to the insertion of 315 logic bombs. For the sake of explanation, we focus on a single protected candidate method, namely the `setButton()` one. This method is invoked each time the user clicks on a blind button in the minesweeper game, and changes the look and feel of the button and its neighbors, as well as the message to prompt to the user, according to the rule of the game (see Figure 4.2). As this method makes decisions, it contains several qualified conditions (e.g., `var == -1`).

Figure 4.4 describes the actual protection workflow, step-by-step. First, NRP parses the `classes.dex` file (Step 2.2) and selects all methods with at least a qualified condition as candidate methods (Step 2.3). NRP adds a logic bomb in any candidate methods and for each qualified condition. Concerning the `setButton()` method, it selects the condition `var == -1` and splits the original code of the corresponding block into two parts, namely `or_code1` and `or_code2`. In Step 2.4, NRP builds up the logic bomb by i) rewriting the if block in the candidate method, and ii) distributing the logic of the bomb among two encrypted files, namely a Java class (i.e., `encr_cl.dex`) and a native C/C++ library (i.e., `nat_encr.so`). Concerning the first point, the qualified condition is translated into a Boolean variable (i.e., `resultMatch`), whose value is calculated by comparing the `var` value with the hash of `-1` encoded as a byte array. In case the value of `resultMatch` is true, the block executes a `decryptAndInvokeMain(...)` method. Such method takes as input the actual name of the encrypted Java class (i.e., `className = AlertController_setButton...`), its binary representation (i.e., `cipherCode`, which is contained in the block), as well as the value of `var` and and the array of arguments (i.e., `args`) required by the main method of the encrypted Java class to execute properly. The invocation of such method leads to the decryption of `cipherCode`, using the value of `var` as decryption key, and to the execution of the main method of the class `AlertController...`. The main method contains the `native_key` to decipher the native library (i.e., `nat_encr.so`), a JNI call and the second part of the original block code (i.e., `or_code2`). The JNI call triggers the execution of a deciphering function in the native library (i.e., `Java_embedded_native_androidxappcompat(...)`, which i) retrieves the encrypted native code from a memory location, ii) decrypts it using the `native_key`, and iii) launches the execution of the decrypted function. This latter function (i.e., `Java_embedded_native_androidxappcompat[...]_KEY(...)`) contains the anti-tampering

check (i.e., `hash_file_in_apk(...)`) that verifies the integrity of the first 100 bytes of the `classes.dex` file. If the integrity is verified, the function executes the first part of the original block code (i.e., `or_code1`). Finally, the JNI call terminates and the main method in the Java class executes the last part of the original block (i.e., `or_code2`).

4.2.6.2 Dismantling NRP bombs

The attacker has to bypass the integrity checks performed by the protected app in the `Java_embedded_native_androidxappcompat[...]_KEY(...)` function to dismantle the NRP logic bombs. To achieve such result, the attacker can either i) override `encr_cl.dex` or `nat_encr.so` for all the triggered logic bombs, by leveraging code deletion and binary patching, or ii) bypass the integrity check performed by the native code, by relying on a replay attack.

Bytecode and Native Code Overriding: The attacker can instrument the code to intercept the invocations to the *decryption functions* both in the Java and the native code, in order to retrieve i) the encrypted code and the decryption key passed as argument to the function, and ii) the decrypted code at the end of that function. Code instrumentation can also be used to change the implementation (or just the return value) of the tamper check function in the native code.

The Java decryption method requires two input values, namely the encrypted code (`param2`) and the decryption key (`param1`), and returns the decrypted code. Figure 4.5 shows the Frida script that instruments the Java decryption method (i.e., `SDCloaderr.decryptAndInvoke(...)`) for a given encrypted code `encr_cl`, represented by `ciphered_byte` (i.e., the `cipherCode` byte array in the `setButton` method), that we will analyze using Jadx [62]). The script begins by selecting (lines 1 and 2) the class (i.e., `embedded.SDCLoader`) and the decryption method

(i.e., `decrypt`) to overload. Frida hooks at the start of the overloaded function and executes the customized code (lines 3 to 19) instead of the original code. The customized code calls the original Java `decrypt` method (line 6) and stores the returned value in a variable. Since the attacker's aim is to get access to a specific decrypted function, the script (lines 7-13) compares the encrypted method passed as the input parameter `param2` with `ciphered_bytes`. If the decryption function is correct, the script writes down the decrypted bytecode (line 16). It is worth noticing that the script also dumps the key inside the `param1` variable. The return value of the overloaded function is the return value of the original decryption function (line 19).

```
1  var SDCLoader = Java.use("embedded.SDCLoader");
2  SDCLoader.decrypt.overload("[B", "[B").implementation = function(param1, param2)
       {
3    var arr_bytes = Java.array('byte', param2);
4    var ciphered_bytes = [107, 30, -22 [..] 18];
5    var found = true;
6    var dexBytesDecrypted = this.decrypt(param1, param2);
7    if(arr_bytes.length == ciphered_bytes.length) {
8     for(var i = 0; i < ciphered_bytes.length; i++) {
9       if(ciphered_bytes[i] != arr_bytes[i]){
10        found = false;
11     break;
12      }
13    }
14     if (found) {
15      console.log("### FOUND");
16      // Write to file dexBytesDecrypted
17     }
18   }
19   return dexBytesDecrypted;
20 }
```

Figure 4.5: Frida script to dump the result of decrypted function.

Concerning the native decryption function, it can be reversed using a decompiler (i.e., Ghidra [63]), and analyzed to detect the decryption function, and to infer the decryption logic (e.g., it is possible to analyze the body of the `Java_embedded_native_androidxappcompat[...]` function). Here, the `decrypt_code` function is responsible for decrypting n bytes of machine code starting from a

pointer to the memory. Such function can be overloaded and dumped through the scripts described in Figure 4.6 and Figure 4.7, which allow retrieving both the plain code and the key, as in the case of the Java decryption function. In detail, the script in Figure 4.6 aims to detect the location in which the target function to dump will be decrypted and executed. It starts by retrieving the `libnative-lib.so` (line 2) - which is the shared object containing all native functions implemented by NRP and then it enumerates each function therein (lines 6-10) to detect the `decrypt_code` functions. Then, the script sets up an interceptor for the function (line 13). Whenever a `decrypt_code` function executes, the script verifies whether the decryption function is the expected one (lines 14-30), by comparing the key passed as a parameter to the `native_key` hardcoded in the `encr_cl.dex` file. If this is the case, the script invokes the `AES_CTR_xcrypt_buffer` function that decrypts the code, write the plain machine code in a writable page, and makes it executable. More specifically, the script in Figure 4.7 attaches Frida to the `AES_CTR_xcrypt_buffer` function (lines 5-14). Then, when the function starts (i.e., OnEnter, lines 15-18), the script retrieves the memory address (i.e., `args[1]`) and the size (i.e., `args[2]`) of the write buffer for the code. After the decryption phase terminates (i.e., OnLeave (from line 19 to line 24), the script checks whether the decrypted code is correct (line 20): in case, the script retrieves the pointer to the starting address (line 21) and dumps `len` bytes (lines 22-24).

```
1  function inspectModule(module) {
2    var m = Process.getModuleByName(module);
3    var targetFunction = true;
4    var functionToBeIntercepted = "";
5    var exports = m.enumerateExports();
6    exports.forEach(function(exp){
7      if (exp.name.indexOf("decrypt_code")>=0){
8        functionToBeIntercepted = exp.name;
9      }
10   });
11
12   // Intercept decrypt_code function
13   Interceptor.attach(m.getExportByName(functionToBeIntercepted), {
14     onEnter: function(args) {
15       var pointer = new NativePointer(args[2]);
16       var buff = pointer.readByteArray(16);
17       buff = new Int8Array(buff);
18       var key = [-63, [...], -9]; // native_key hardcoded in encr_cl.dex
19       targetFunction = true;
20       for(var i = 0; i < buff.length; ++i){
21         if(buff[i] != key[i]){
22           targetFunction = false;
23           break;
24         }
25       }
26       if(targetFunction) {
27      console.log("##### FOUND");
28       }
29     }
30   });
31
32   // Intercept AES_CTR_xcrypt_buffer function
33   [...]
34 }
```

Figure 4.6: Frida script that searches for the target `decrypt_code` function.

```
1  function inspectModule(module) {
2    [...]
3    // Intercept AES_CTR_xcrypt_buffer function
4    // Dump the return bytes
5    functionToBeIntercepted = "";
6    exports = m.enumerateExports();
7    exports.forEach(function(exp){
8      if (exp.name.indexOf("AES_CTR_xcrypt_buffer")>=0){
9        functionToBeIntercepted = exp.name;
10     }
11   });
12   var outAddr = "";
13   var len = 0;
14   Interceptor.attach(m.getExportByName(functionToBeIntercepted), {}
15     onEnter: function(args) {
16       outAddr= args[1];
17       len = parseInt(args[2]);
18     },
19     onLeave: function(retval) {
20       if(targetFunction) {
21         var pointer = new NativePointer(outAddr);
22         var buff = pointer.readByteArray(len);
23         // DUMP buff
24         send("Readed byte", buff);
25       }
26     }
27   });
28 }
```

Figure 4.7: Frida script that dumps the assembly code decrypted by the native function.

The decrypted code is a hex string that encodes the x86 assembly instructions. Such instructions can be extracted using a disassembler. The analysis of the disassembled instructions allows detecting a hardcoded tamper check on the `classes.dex` file

```
cmp eax, 0xb543c475
```

where $0xb543c475$ is the checksum of the original `classes.dex` file of Antimine.

Once the original codes and the decryption keys are recovered, an attacker can easily bypass the tamper check by overriding the hardcoded value with the one of the repackaged app, and then re-encrypting the assembly code and substituting the original code in the *.so* file. As an alternative, the attacker can modify the decrypted `nat_enc.so` and the `encr_cl.dex` to execute `op_code1` in Java rather than in the native code, and remove the JNI call that would trigger the tamper check.

Figure 4.8: The original (top side) and the tampered (bottom side) library file.

Bypass the integrity check: An alternative attack to bypass the tamper check is to leverage a reply attack by modifying the file on which the tamper check is executed. This attack can be carried out since the path of the checked file is hardcoded in the native library in clear text.

We carried out this attack as follows: we added a new file (i.e., `cla.dex`) to the *apk*, that has the same content of the original `classes.dex` of Antimine. Then, we manually patched the hardcoded path in the `elf` file to point to `cla.dex` instead of the `classes.dex` which is modified by the attacker before repackaging. In this way, the tamper check passes as it calculates the expected value, despite the app content has been modified. Figure 4.8 shows the original library (top side), and the tampered one (bottom side) in which the hardcoded path is modified to point to *cla.dex*.

Previous attacks allowed to successfully bypass the NRP protection on Antimine. The repackaged version of Antimine, with all the 315 bombs de-activated, is available at:

`https://www.csec.it/projects/antimine-800011-protected-and-repacked.apk`

for further studies, analysis and testing activities.

Chapter 5

ARMANDroid: Anti-Repackaging through Multi-pattern Anti-tampering checks based on Native Detection

Summary

In this chapter, we discuss the requirements that a next-generation anti-repackaging tool should meet. Then, we propose a solution that fulfills such requirements as well as its implementation in a tool named ARMANDroid.

5.1 Methodology

In this section, we discuss a methodology that aims to overcome the limitation of the current SOTA, by addressing the following challenges, we already pointed before, namely:

1. **Use multiple patterns** to disseminate and hide logic bombs in the app. Moreover, several heterogeneous anti-tampering controls must be distributed throughout the app.

2. **Rely on native code** to hide both logic bombs and their triggering condition, as reversing and instrumenting native code is more complicated than bytecode.

3. **Optimize the pervasiveness of the logic bombs**, in terms of effectiveness. Some approaches implement few controls in the name of performance, while others embed a lot of bombs which are not actually triggered at runtime. In this respect, there is the need for a more effective deployment of logic bombs, also at the cost of higher time and space overheads.

4. **Honeypot bombs**. Proper fake bombs can be added to disguise the attacker to remove them. The removal of such bombs should trigger the real ones and lead the app to crash. This idea has been proposed in literature but never actually applied in any proposals, where the focus is more on generating stealthy logic bombs.

5. **Hiding anti-repackaging protection**. It is fundamental to improve the stealthiness of anti-repackaging to make hard for the attacker to understand that the reverse-engineered app actually contains some protection. Therefore, obfuscating anti-repackaging and anti-tampering controls may help.

5.1.1 Multi-Patterning & Anti-Tampering Checks

The robustness of an anti-tampering technique strongly relies on the distribution and the hiding of logic bombs. Unfortunately, current proposals rely on a single

pattern, in which any qualified condition is systematically substituted with some hash value and the body code encrypted according to the `const` value. Such pattern can be straightforwardly detected by the attacker through static analysis based on pattern matching. As a consequence, current anti-repackaging proposals are more concerned on spreading a high number of logic bombs rather than hiding the portion on code in which an anti-tampering check can be contained. From an attacker's point of view, the knowledge of this well-known pattern allows revealing the potential location of the anti-tampering detection nodes (i.e., all branching containing hash values in their condition) and automate their removal.

We argue that adopting a multi-pattern distribution of logic bombs can increase the effort of the attacker and reduce the likelihood to removing the whole set of logic bombs.

Multiple logic bomb patterns: As introduced above, an attacker can leverage a single combination of function calls to detect the hash-based logic bombs. To counteract this threat, we build logic bombs according to the classical pattern, but we modify their inner code. In this way, even if an attacker detects several logic bombs, she is not able to decrypt them and recover the original code with a single procedure because each bomb has a different inner structure. More in detail, we adopt six different types of logic bombs:

1. **Honeypots** are fake logic bomb, which do not contain any anti-tampering check. Honeypots aim at tricking the attacker to spend time and effort to uselessly decrypt them.

2. **Java bombs** that contain one or more Java anti-tampering checks.

3. **Native-key bombs** are made by encrypted inner code where the decryption key is the return value of an anti-tampering check that is implemented

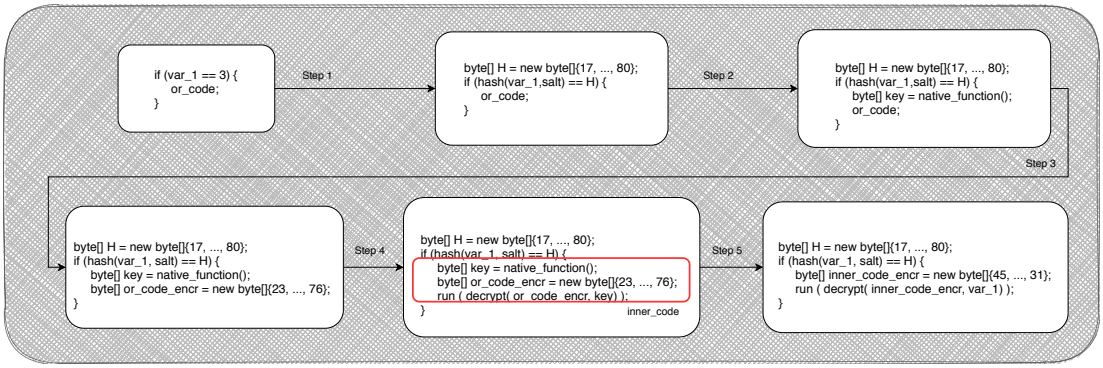


Figure 5.1: Creation process of a native key bomb

in the native code. Figure 5.1 shows the creation process of this type of bomb. Here, the qualified condition is transformed into the form of the logic bomb (Step 1). Inside the qualified condition a native call is embedded (Step 2) and its return value is used to encrypt the `or_code` (Step 3). Then, proper instructions to decrypt and run `or_code_encr` are added (Step 4). In this way, the `inner_code` contains the native invocation, the decryption process and the invocation of `or_code`. Finally, the `inner_code` is encrypted with the `const` value in the qualified condition (Step 5).

4. **Native AT bomb** implements an anti-tampering check inside the native code. Differently from the previous cases, if the native AT check fails the normal app workflow is broken (i.e. an exception is thrown). In future versions, different actions can be implemented once a tampering is detected.

5. **Java AT & Native key bomb** are bomb analogous to the previous type, but `or_code` contains at least one Java anti-tampering check.

6. **Java & Native AT bombs** contain both Java and native anti-tampering checks.

Beyond logic bombs: In addition to logic bombs, it is necessary to embed heterogeneous anti-tampering controls throughout the app. We inserted some detection nodes which check the application integrity. These nodes, called **Java AT**, are spread through the Java bytecode. If all detection nodes rely on the same anti-tampering control, an attacker would bypass it to create a working repackaged app. In our work, each anti-tampering node checks different things, and it detects a *specific type* of repackaging. For example, file integrity checks detect a repackaging if and only if the verified file is changed during the repackaging process. More in general, checking the signature can potentially detect all repackaging: an attacker have to re-sign the repackaged version of the app with his private key (that is different from the developer one). Furthermore, these checks can be implemented both in Java and native code, in order to increase the amount of work an attacker need to carry out to circumvent these controls.

Deployment of logic bombs: Another important aspect is where logic bombs are deployed and how anti-tampering checks are inserted. In our previous analysis, we notice that some logic bombs could be decrypted only by relying on reverse engineering and open-source information. An example is a logic bomb inserted inside an external open-source library. From the point of view of the defensive technique, these bombs do not add security but only increase the time overhead in the repackaging process. One of the powerful technique that an attacker can use in the repackaging process is *code instrumentation and injection*[1], which allow dynamically modification of the code, intercepting function calls, and changing input or return values. To make the reversing process harder, an anti-tampering control (both in Java and native code) is inserted replicating its bytecode (or assembly code) each time. In this way, we do not rely on a static anti-tampering function that is called from each detection node, despite there is and higher size

[1] discussed in 4.1.6

overhead.

5.1.2 Rely on native code

The reverse engineering and the instrumentation process of native code (assembly code) are more complicated than Java bytecode. Due to this, interacting with native code can increase the difficulty of the repackaging process. To interact with native functions, a Java class has to load the shared object (usually with the directive `System.loadLibrary` [64]) and declare a method with the `native` keyword (this method tells the virtual machine that the function is implemented in the shared library). Moreover, there are several constraints to be respected (e.g., the name of a native function) when dealing with JNI. Due to this, a native function call could be detected easily. To avoid that an attacker easily removes the call to a native anti-tampering check (through *code deletion* 4.1.6), the Java code should be dependent on the result of the native function. To this aim, we identified two kinds of native invocation: **native key** and **native AT** (described above in Section 5.1.1)

5.1.3 Hiding protection

Logic bombs are a first attempt to hide detection nodes, in which one or more anti-tampering controls can be embedded. Furthermore, logic bombs can be nestedly hidden, i.e., the content of a logic bomb hides another encrypted payload. In this way, an attacker has to decrypt all nested logic bombs to recover and remove the anti-tampering checks. As we mention above, logic bombs and anti-tampering controls have common patterns that help an attacker in the detection of anti-tampering controls. Obfuscation can be applied to the protected app to hide common features and make hard the repackaging process for an attacker (i.e., detection of logic bombs or anti-tampering checks). In our work, we implemented

this solution by relying on Obfuscapk [23].

In conclusion, these considerations lead us to create a protection scheme in which i) there are multiple patterns for logic bombs, ii) logic bombs are inserted through the code with specific and semi-stochastic criteria, iii) other heterogeneous detection nodes are inserted *randomly* in the code and not only inside the logic bombs and iv) there are multiple anti-tampering checks (i.e., signature check, file integrity, package name check) in both Java and native code.

5.1.4 Runtime behavior

At runtime, the Java anti-tampering spread out the Java code and the logic bombs are executed. The code in logic bombs is properly decrypted only if the value of the variable (`var_1`) in the qualifying condition is equal to `const`. Then, if the logic bomb contains at least one anti-tampering function, a tampering check is executed. These functions calculate some digests and compare them with the stored one: tampering is detected whether one of such value differs from the stored one. In case tampering is detected, the app crashes and throws an exception. The type of the exception depends on which anti-tampering is executed.

As an example, for native key logic bombs, the native anti-tampering returns the key to decrypt the next code. If tampering is detected, the native function returns a wrong value (i.e., return a wrong decryption key). In this scenario, the *AES* decryption function fails, throwing an exception.

5.2 Implementation

We implemented our anti-repacking methodology in a prototype tool called ARMANDroid. The tool is mainly written in Java and relies on some supporting

frameworks, e.g. Axml[1] and FlowDroid[2] to unpack, analyze and repack the apk files. Furthermore, ARMANDroid relies on the Soot framework [65] to transform the Java bytecode in an intermediate representation, called Jimple, that facilitates the injection of the AT protection mechanisms.

ARMANDroid requires four input parameters, i.e., i) the percentage of Java AT (P_{Java_AT}), ii) the percentage of native key bombs (P_{native_key}), iii) the percentage of native AT bombs (P_{native_AT}) and iv) the package name of the app to protect. The first three percentage values express the probability of ARMANDroid to insert, respectively, a Java AT, a Native Key bomb, and a Native AT bomb. Each percentage value is unrelated with the other values, and their sum does not need to be 1 because they are independent. The latter parameter allows to change the package in which the tool injects the AT protections controls. If this parameter is set to `none`, the tool protect all Java classes contained into the application.

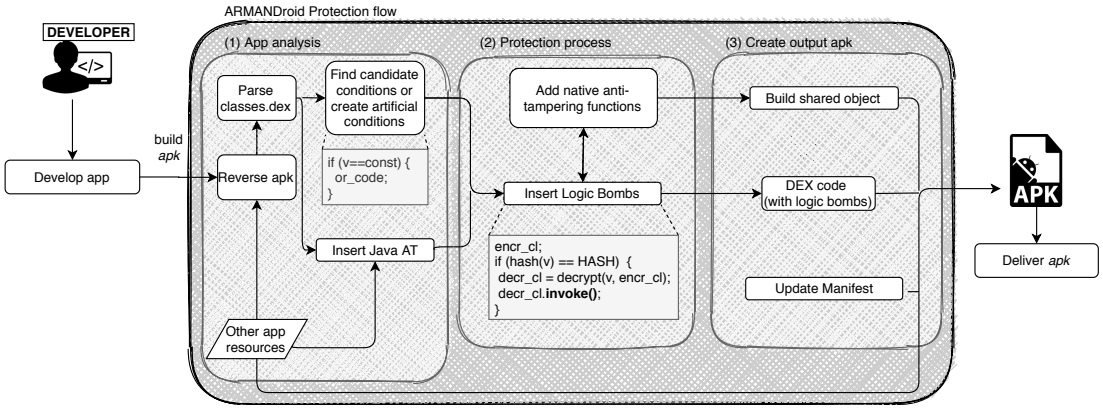


Figure 5.2: ARMANDroid AT protection workflow.

The protection process can be divided into 3 macro-steps (see Figure 5.2):

1. *App Analysis.* ARMANDroid unpacks the *apk* package to extract the

[1]https://github.com/Sable/axml
[2]https://github.com/secure-software-engineering/FlowDroid

`classes.dex` file(s). The bytecode is then translated into the Jimple intermediate language and analyzed using the Soot framework [65]. The analysis inspects the app code to find candidate QC. Furthermore, in this phase, Java anti-tampering controls are embedded into the bytecode.

2. *Code transformation.* For each QC, the tool inserts - according to the input probabilities - an anti-tampering protection, i.e. one of the six logic bombs defined in Section 5.1.1. To do so, the tool can introduce additional native code.

3. *Create output apk.* Finally, ARMANDroid transforms back the Jimple representation into bytecode, adds the native libraries, updates the `Android Manifest.xml` file, and repack the application, that can be signed by the app developer and redistributed in the wild (i.e., uploaded on the Google Play Store).

The rest of this section details the implementation of each step.

5.2.1 App Analysis

In this phase, the tool unpacks and extracts from the apk file the classes.dex file(s) and transforms the Java bytecode in Jimple using Soot. Then, ARMANDroid iterates over all instructions to find QCs and add Java anti-tampering checks. The results of this phase are i) the transformation tree containing all candidate QCs and ii) the bytecode that can contains several Java anti-tampering.

Searching for QC: The parsing phase scans the Jimple code to detect candidate blocks where logic bombs can be created. These blocks must match the pattern depicted in Figure 5.3, i.e., an if statement with an equality comparison between a variable (`var_1`) and a constant value (`const`).

```
1       [...]
2       if (var_1 == const) {
3           or_code;
4       }
5       [...]
```

Figure 5.3: Expected form of a logic bomb

In the current implementation, ARMANDroid considers two types of candidate QCs: i) if statement and ii) switch-case statement. For the latter case, each case of the switch is a candidate QC.

Moreover, for each candidate block, the tool also computes the number of instructions that can be encrypted in the bomb, namely the *encryption range*. From a technical standpoint, the bomb can encrypt a straight line of instructions without branches until the next label (i.e., the next basic block). Finally, ARMANDroid creates a list of *transformation trees* which contains each candidate blocks.

Add Java anti-tampering: During the analysis, the tool embeds one or more Java anti-tampering controls into the code: an AT will be added before each instruction in a method with a probability P_{Java_AT}, as defined in Figure 5.4.

5.2.2 Code transformation

The transformation process, depicted in Figure 5.5, is carried out for each candidate QC, extracted from the leaves to the root of the transformation trees, and it is composed of six sub-steps.

In Step 2.1, the qualified condition is transformed into the encrypted form. In details, the condition $var\_1 == const$ is transformed into $hash(var\_1, salt) == H_{const}$, where the `salt` is a random value and H_{const} is the precomputed hash value of `const`.

```
1       Iterator<Unit> iter = units.snapshotIterator();
2       Unit currentUnit = iter.hasNext() ? iter.next() : null;
3       Unit nextUnit = null;
4       while (iter.hasNext()) {
5           nextUnit = iter.next();
6           Stmt stmt = (Stmt) currentUnit;
7
8           [...]
9
10          if (!(stmt instanceof IdentityStmt) &&
11                  random.nextInt(100) >= (100-P_JAVA_AT)) {
12              // add Java AT before stmt
13              [...]
14          }
15          currentUnit = nextUnit;
16      }
```

Figure 5.4: Add Java anti-tampering snippet

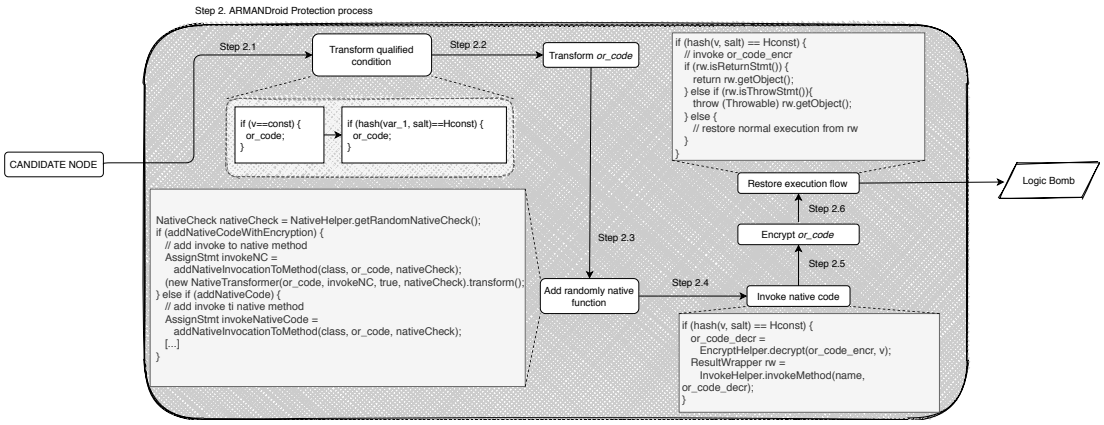


Figure 5.5: Transformation of a candidate node into a logic bomb

Step 2.2 applies some transformations to `or_code` to ensure the correct execution flow of the original method. In particular, we identify four possible situations:

- *return statement:* the original method will terminate returning the correct object.
- *goto statement:* the original method will continue from a specific label.
- *throw statement:* the original method will terminate throwing an exception.

- *normal flow:* the original method will continue the execution from the next instruction.

To handle these situations, we embed a new class `ResultWrapper` (similar to the solution proposed in [39]) into the application.

After the previous transformations, in Step 2.3, before encrypting the `or_code`, a native function call can be inserted with a probability of $P_{native} = P_{native\_key} \cup P_{native\_AT \mid not\ native\_key}$, which corresponds to the join of the chance to create a logic bomb with *native key*, P_{native_key}, and the chance of native anti-tampering under the condition that a native key bomb is not created (see Section 5.1.1). During this step, the added native functions are stored into `NativeHelper` class (Step 2.4).

In Step 2.5, ARMANDroid encrypts `or_code` in the *encryption range* using AES-128 and the result is saved in an encoded string using *base*64 (`or_code_encr`). The encryption/decryption key is derived from the value of `const`. A new variable containing the encoded string is then inserted inside the original method. Then, we add the necessary instructions to decrypt and invoke the cipher basic block. To this aim, we embedded two other classes: `EncryptHelper` and `InvokeHelper`. The first one is responsible to decrypt the ciphered string given the decryption key. The latter class is responsible to load a byte array (corresponding to the bytecode result from the decryption of the `or_code_encr`) into the memory and invoke the *execute* method. In Step 2.6, the tool recovers the correct execution flow based on the `ResultWrapper` object.

At the end of this phase, Java and native anti-tampering as well as logic bombs (of the six types described in Section 5.1.1) are spread across the bytecode.

5.2.3 Create output apk

The last step enables the generation of a working protected apk file. First, the Jimple representation of code is transformed back into the Java bytecode. Then, the `NativeHelper` class generates a new *C* file with the body of the native functions added in the previous phase. Then, the tool compiles a set of shared libraries (i.e., `x86`, `x86-64`, ...) and embeds them into the output apk. The `AndroidManifest.xml` file of the application is updated in order to make it consistent to the transformation carried out into the bytecode. For example, if a new application class is created, the *android:name* property of *application* tag is added to the manifest, [66]. Then, to add one more layer of complexity, ARMANDroid obfuscates the protected app using Obfuscapk [23]. Finally, the output apk is signed with the developer private key and ready to be deployed.

5.3 Experimental

We empirically assessed the reliability of the proposed methodology by systematically analyzing 554 apps downloaded from F-Droid [31] and APKpure [67] between August and September 2020. The experiments were conducted on a laptop running Ubuntu 18.04 with 8GB of RAM and a dual core processor.

5.3.1 Experimental results

The tool was able to successfully process 478 out of 554 apps (i.e., 86%) in nearly 5 hours (274 minutes). In the experimental evaluation we setted $P_{Java\_AT} = 10\%$, $P_{native\_encryption} = 40\%$, $P_{native\_AT} = 60\%$, and the *ApplicationId* as the base package name.

The analysis of the remaining 76 apps failed due to a Soot library's well-known

	Processing Time (sec)	Size Overhead (MB)
min	10	0
max	280	12
avg	33	2.7

Table 5.1: Min, max and avg values of processing time and size overhead.

issue such as[1] or tool bugs, which results in a crash during the analysis of the Java bytecode. In addition, some applications (59 of the 478 apps protected) do not include any anti-tampering protection mechanism due to the fact the package name inside the `AndroidManifest.xml` file contains few lines of Java bytecode (e.g., hybrid app or obfuscated app). For this reason, the tool does not find candidate QCs suitable to embed logic bombs. For these apps, it would have been necessary to change the input package name parameter properly.

Tables 5.1 and 5.2 contain the results of ARMANDroid of the remaining 420 apps in terms of minimum, maximum and average values for computation details and protection mechanisms.

	Honeypot	Java bomb	Native key bomb	Native AT bomb	Java AT & Native key bomb	Java AT & Native AT bomb	Nesting level
min	0	0	0	0	0	0	1
max	258	50	508	561	75	93	6
avg	30	4	53	51	6	7	2

Table 5.2: Min, max and avg statistic of the introduced protection mechanisms.

Space Overhead. The total space overhead is due to two factors: a fixed cost (calculated as less than 1MB), and a dynamic cost related to the number of added detection nodes. In particular, we noticed that the number of Java AT and Native Key bombs are directly proportional to the space overhead since they

[1]https://github.com/soot-oss/soot/issues/1413

contribute to increase the payload size of each method. Furthermore, since native functions are compiled into a shared object for each architecture and embedded into the application, the number of native methods is directly proportional to space overhead. Finally, it is worth noticing that the average overhead size of protected APK in the dataset is limited (only 2.7 MB).

Time Overhead. Concerning performance, ARMANDroid took 4 hours and 34 minutes for protecting all the apps, with an average of 33 seconds for each app, on a mid-level laptop, thereby suggesting that the approach is viable. The building time is mostly under 2 minutes, even for the app with a lot of qualified conitions.

Achieved Protection Level. The results show that ARMANDroid successfully includes a fair amount of logic bombs (i.e., more than 150) distributed among the six possible combinations. Furthermore, the average nesting level of those bombs is equal to two, thus enhancing the difficulty for their defusal. Nevertheless, there are a few exceptions. Indeed, for apps with a limited set of QCs, the tool may introduce a small set of bombs that do not cover all the six possibilities. To this aim, the computation of the min value brings to a raw full of zero. However, it is worth noticing that such a problem can be overcome by using a more permissive package name to cover, for instance, the entire app and not only the $ApplicationID$ package.

5.3.2 Parameters Tuning

Since ARMANDroid performs the code transformation according to the probabilities of injection, we conducted a tuning tests to detect the on-average best combination of parameters to ensure a reasonable trade-off between the space

% Java AT	% Native Key	% Native AT	Avg. Java AT	Avg. LB	Avg. Nesting	Avg. Protection time (s)	Avg. Size overhead (MB)
5	90	20	301	300	2.6	46	4.2
	70	40	303	303	2.7	46	4.0
	40	70	307	306	2.3	47	4.1
	20	90	307	308	2.3	47	4.0
10	90	20	670	305	2.6	50	4.1
	70	40	661	309	2.6	52	4.2
	40	70	629	296	2.5	46	3.8
	20	90	607	295	2.2	44	3.9
20	90	20	1328	281	2.6	48	4.4
	70	40	1230	262	2.5	48	4.7
	40	70	1306	285	2.0	50	4.6
	20	90	1311	287	2.3	51	4.5

Table 5.3: Tuning of ARMANDroid input parameters according to the resulting protection level, computation and size overhead.

and time overhead and the protection level. To do so, we computed twelve different combinations of $P_{Java\_AT}, P_{native\_key}$, P_{native_AT}. We used a test-bed of 50 apps to evaluate the resulting average number of inserted Java AT, logic bombs, the average achieved nesting level of these bombs, and the average computation values (i.e., time of protection process and size overhead). For the experiments, we used the $ApplicationID$ of the app as package name to be processed. The results of the assessment, reported in Table 5.3, lead to some considerations.

In all cases, the average nesting level of bombs is greater than 2, while the average number of included ARMANDroid detection nodes (i.e, the sum of Java AT and the six types of logic bomb) ranges from 601 to 1609. Such results ensure that - despite the input parameters - all the apps are equipped with complex and hard-to-defuse bombs.

Regarding the space overhead, the results underline that the variation of input parameters does not affect significantly the size of the protected apps: most applications have a size overhead of less than 6MB, with an average value around 4MB. In the worst case, the size overhead is around 12MB, where an average of

2790 Java anti-tampering, 1490 logic bombs with 1290 native calls are inserted. Moreover, the maximum nesting level is 5. All in all, it is worth noticing that all the apps do not violate the size limit of 100MB, imposed by several application stores, such as Google Play [68].

Finally, the experiments underline that the most in the processing time is due to the percentage of Java anti-tampering (P_{Java_AT}). In details, the average protection time is directly proportional to the number of embedded Java ATs as those detection nodes introduce additional bytecode that will be processed. On the contrary, the percentage of native key (P_{native_key}) and native AT (P_{native_AT}) do not relevant affect the compilation time. Changing these values, the percentages of each logic bomb type is altered, but each application has the same number of candidate qualified condition.

5.3.3 Effectiveness

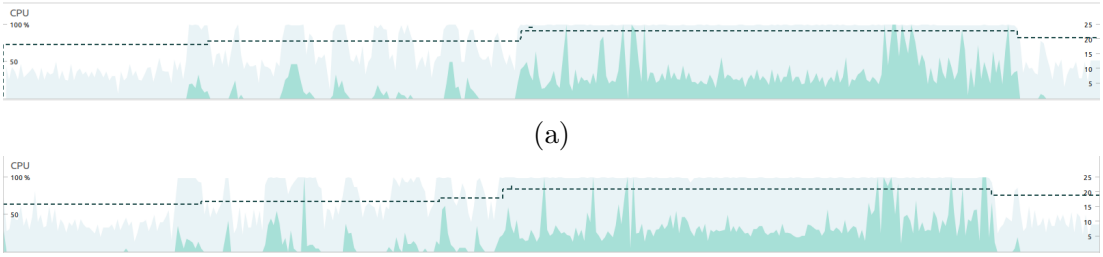
To dynamically test the applications after the anti-repackaging process, we randomly selected twenty apps from the dataset and we tested at runtime both the original and the protected ones. For the testing phase we used an emulated Android 8.0 phone equipped with a dual core processor, 2GB of RAM, and the latest version of Google Play Services.

Firstly, we used Monkey [32] command-line tool to generate pseudo-random input events on the emulator for five minutes to detect possible crashes or exception of the protected app. All the selected twenty protected apps behave as expected and no new exceptions were thrown.

Then, we conduct the profiling of apps in terms of CPU usage, memory usage, and energy consumption. For those tests, we used the Android Profiler[1] tool to track both the original and the protected application. The collected data were

[1]https://developer.android.com/studio/profile/android-profiler

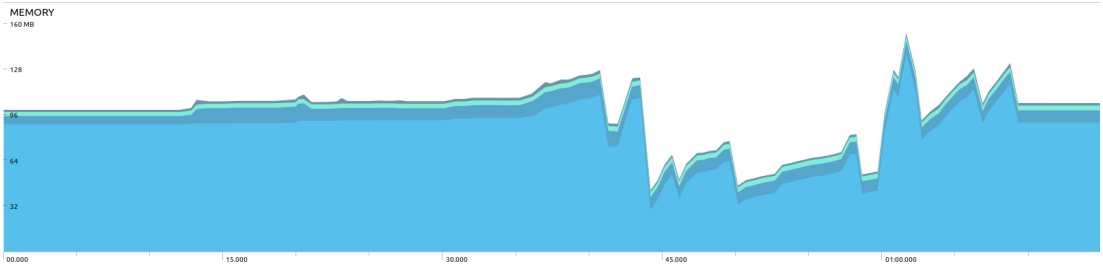
exported and compared to evaluate the run-time overhead introduced by the ARMANDroid protection. The results demonstrated that the overhead of CPU, memory, and energy in the protected apps is negligible and the user does not notice any difference.



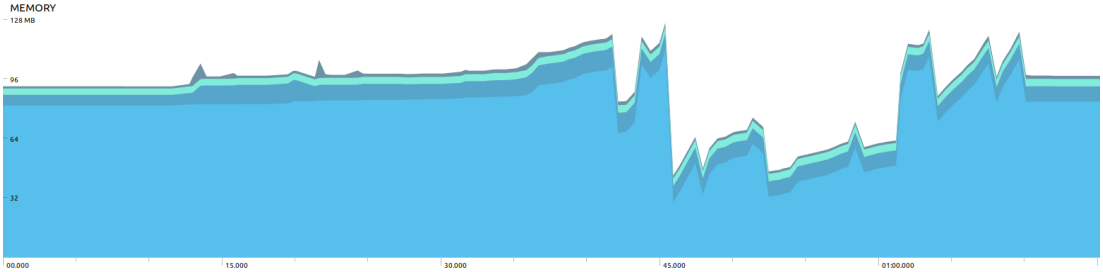
(a)

(b)

Figure 5.6: Comparison between the CPU usage of (a) original and (b) protected APK.



(a)



(b)

Figure 5.7: Comparison between the memory consumption of (a) original and (b) protected APK.

For instance, Figures 5.7, 5.6, 5.8 report the CPU usage, the memory usage

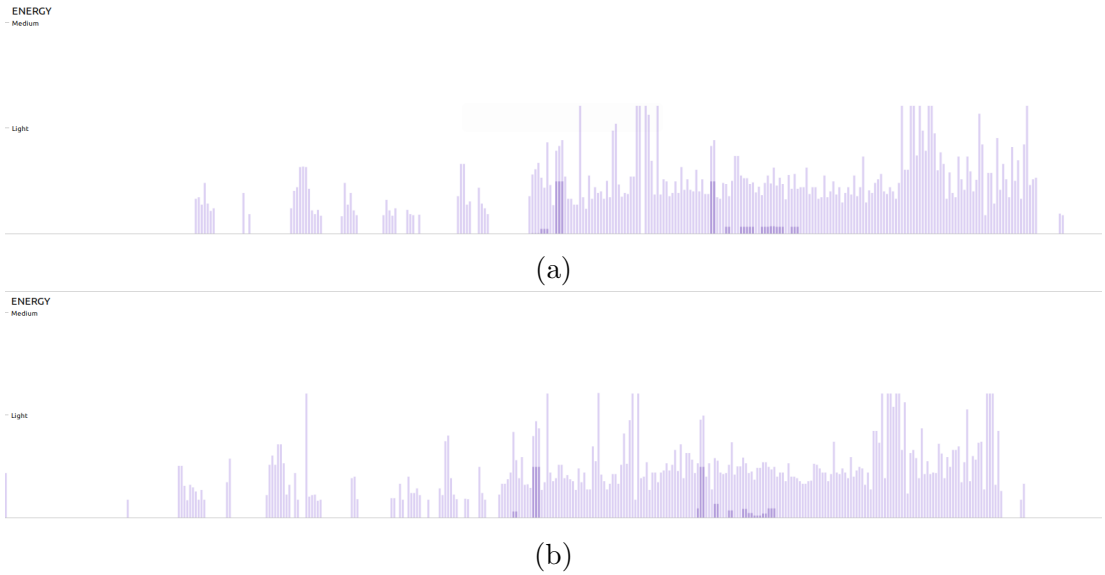


Figure 5.8: Comparison between the energy consumption of (a) original and (b) protected APK.

and the energy consumption between the original (a) and the protected version (b) of *org.twistedappdeveloper.statocovid19italia.apk* application. Both apps were executed with the same input sequences for a one-minute and thirty seconds. During this execution time, the protected app triggered 164 Java anti-tampering and 15 logic bombs with 8 native anti-tampering checks. Nevertheless, the waveforms of each graph have no visible deviations.

Chapter 6

Conclusions

In the first part of this thesis, we discussed the state of the art of anti-repackaging techniques, unveiling their limitations and the way to attack them. Such techniques are far from being robust against the available attacking vectors. We identified the guidelines to improve the reliability of anti-repackaging techniques. Then, we proposed ARMANDroid, an anti-repackaging tool which aims to overcome current limitations. The evaluation shows that the protected applications are equipped with several detection nodes and the performance impact is negligible.

We focused on Android OS, but, as future enhancement, this approach can be extended also to iOS. Furthermore, our tool can be improved by adding other anti-tampering functions or finding more candidate qualified conditions.

With this thesis, we hope to be able to help Android App developers against the repackaging threat.

We submitted the results of this thesis to a Q1 journal (i.e., Computers & Security) for consideration. The paper is a review paper, in which we submitted an extended version of the analysis of the SOTA (Chapters 1 to 4) and the guidelines for next-generation anti-repackaging solutions. We are also working on a second submission to another Q1 journal (i.e., Pervasive and Mobile Comput-

ing), in which we will present our methodology and an extended assessment of ARMANDroid (i.e., on a higher number - 5K - of mobile apps taken from the Google Play Store), to prove the viability of the proposal in the wild.

Appendix A

UML Class Diagram

In this appendix we show the class diagram of the developed tool. We propose a diagram for each package to make it clearer. The project includes:

- ***Main.java*:** It is the entrypoint of the tool. It is responsible to parse input arguments, setup specific parameters and start the transformation process.
- **Package *util*:** The classes of this package support different phases of the transformation process.
- **Package *sootTransform*:** The classes of this package are responsible to find the candidate qualified condition in every method, perform the transformation into logic bombs and insert Java and native anti-tampering.
- **Package** *models*: It contains all models of object created. There are tree embedded packages:
  - *javaChecks:* It contains the source code of Java anti-tampering.
  - *nativeChecks:* It contains classes that wrap native anti-tampering into Java code.
  - *exception:* It contains custom exceptions.

- **Package *embedded*:** It contains all classes that will be embedded into the protected app.

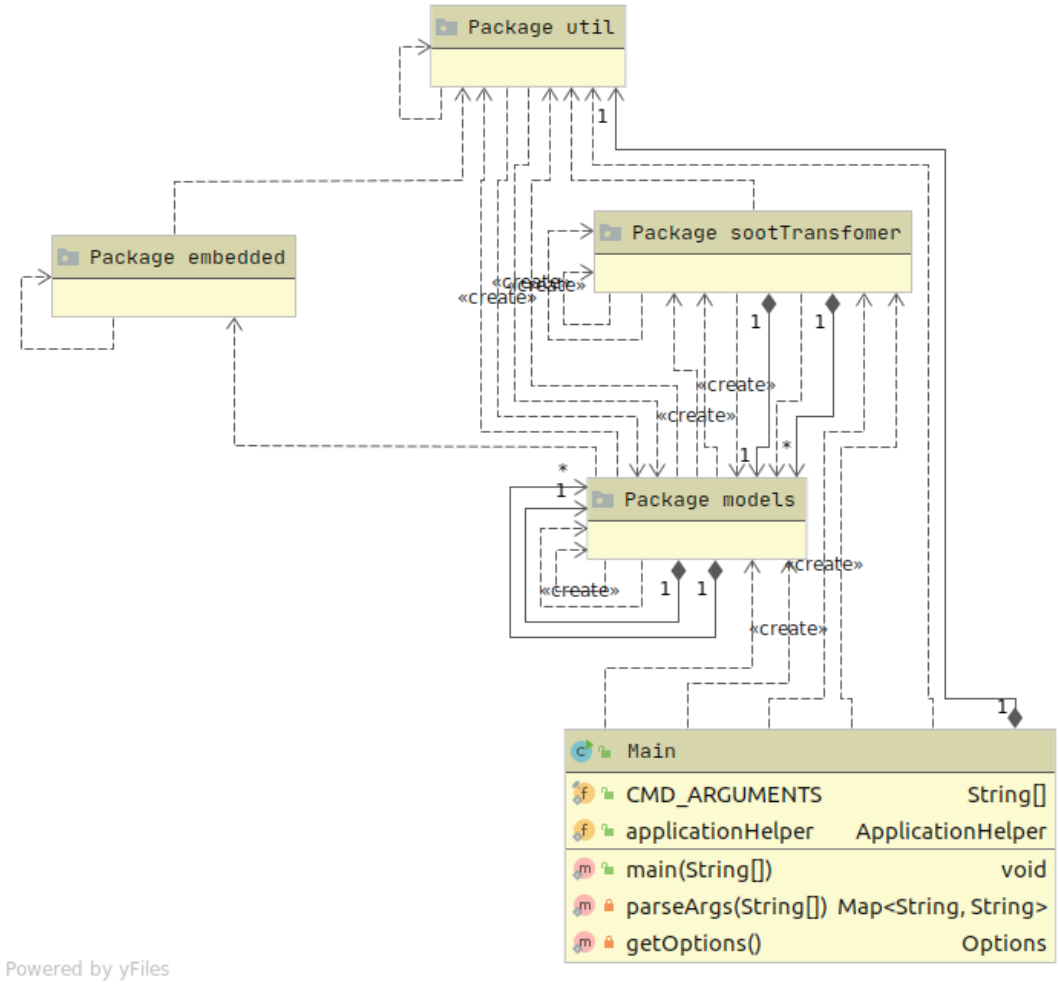


Figure A.1: Main class

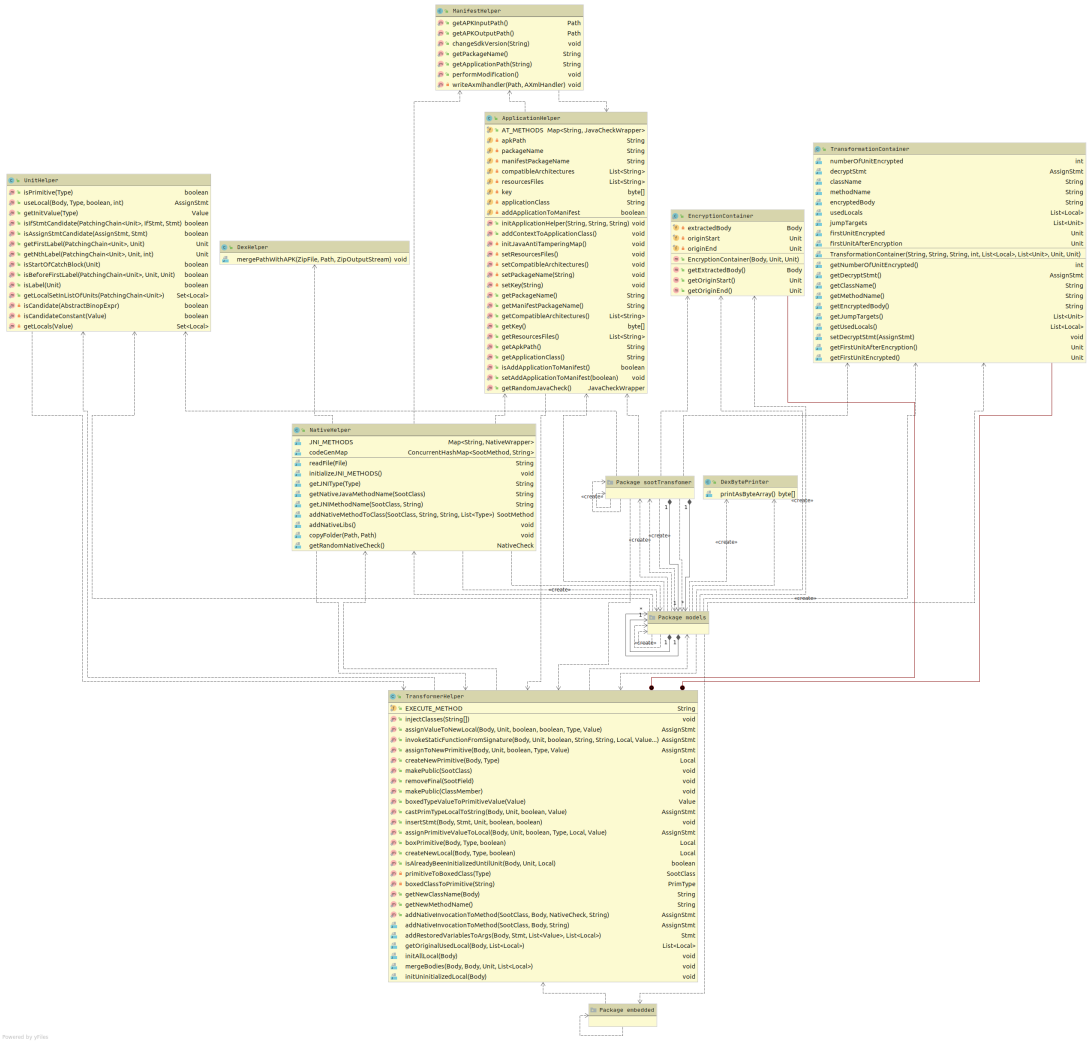


Figure A.2: Package util

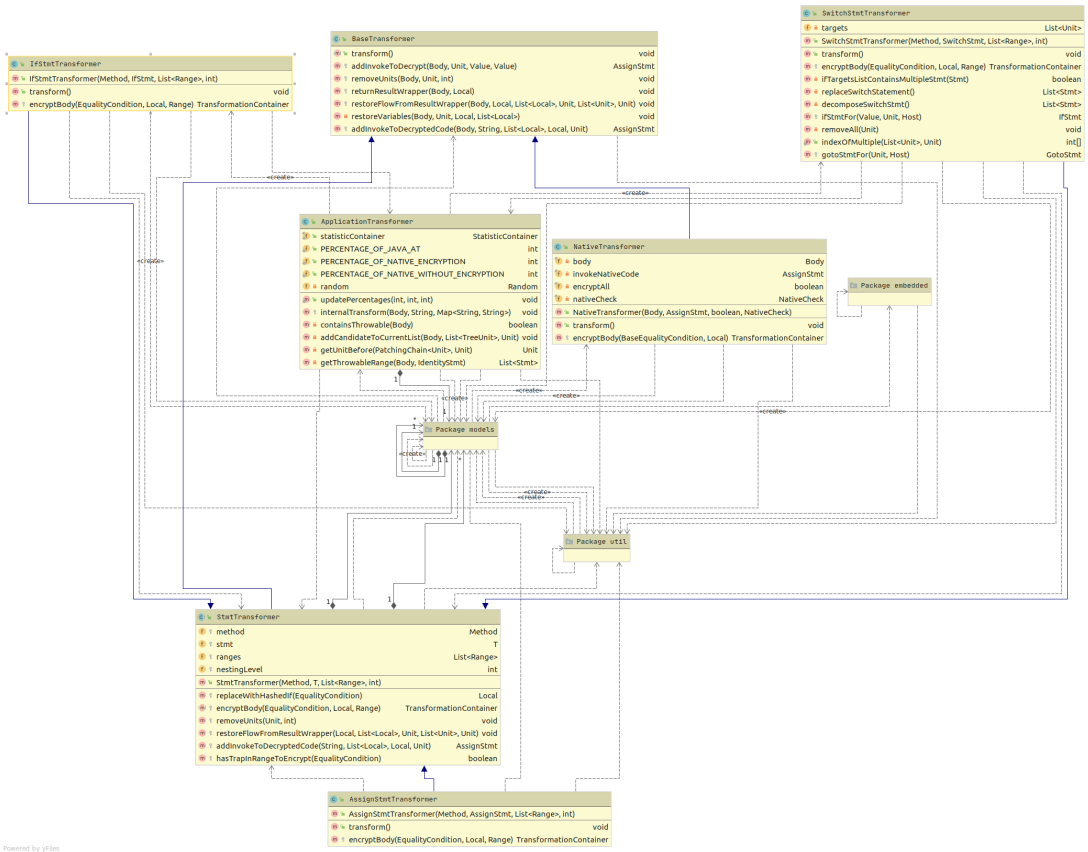


Figure A.3: Package sootTransformer

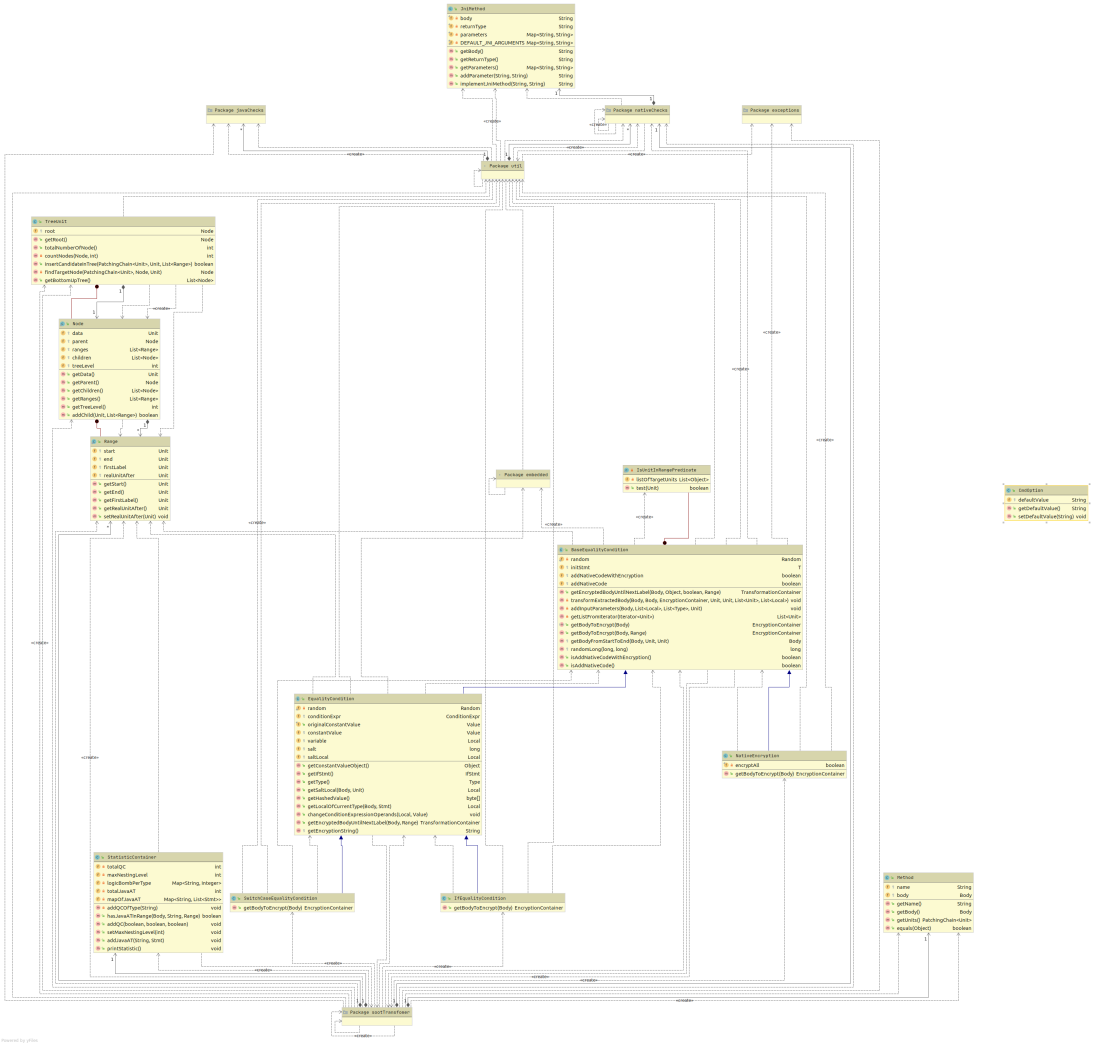


Figure A.4: Package models

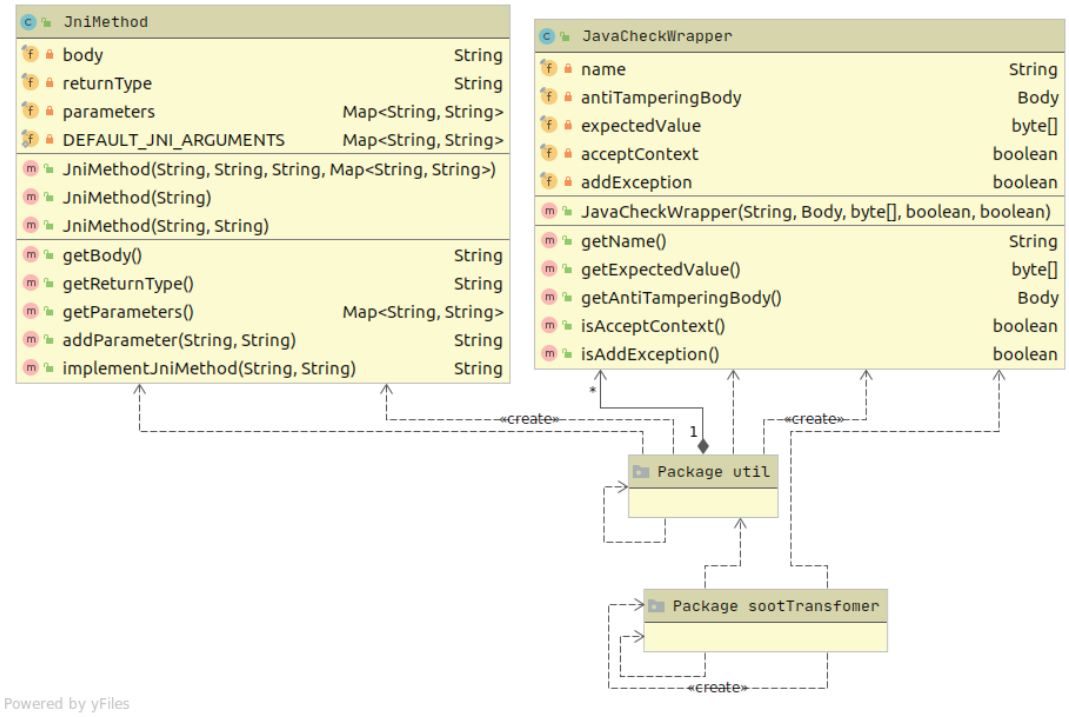


Figure A.5: Package models.javaChecks

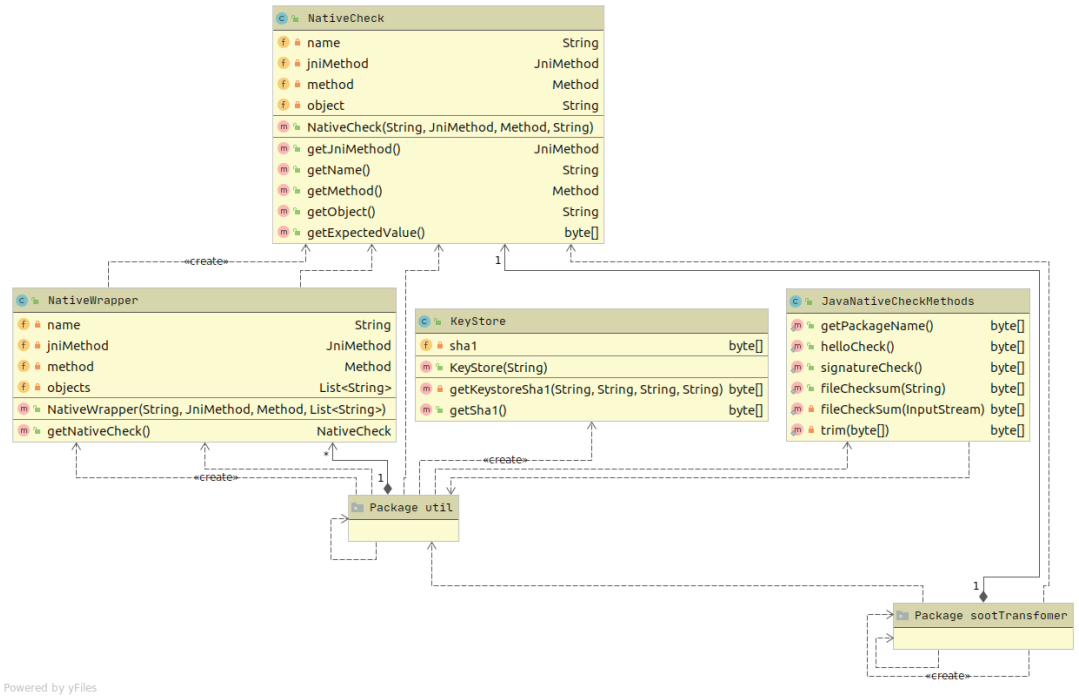


Figure A.6: Package models.nativeChecks

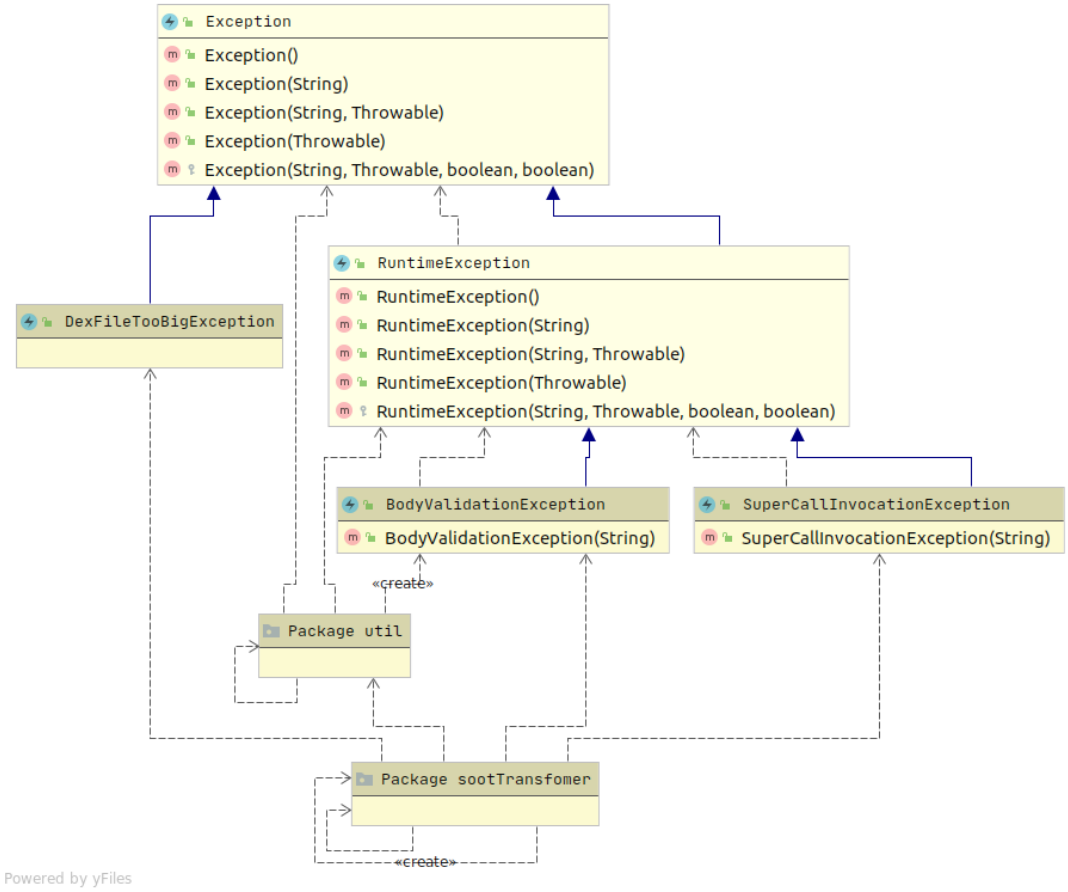


Figure A.7: Package models.exceptions

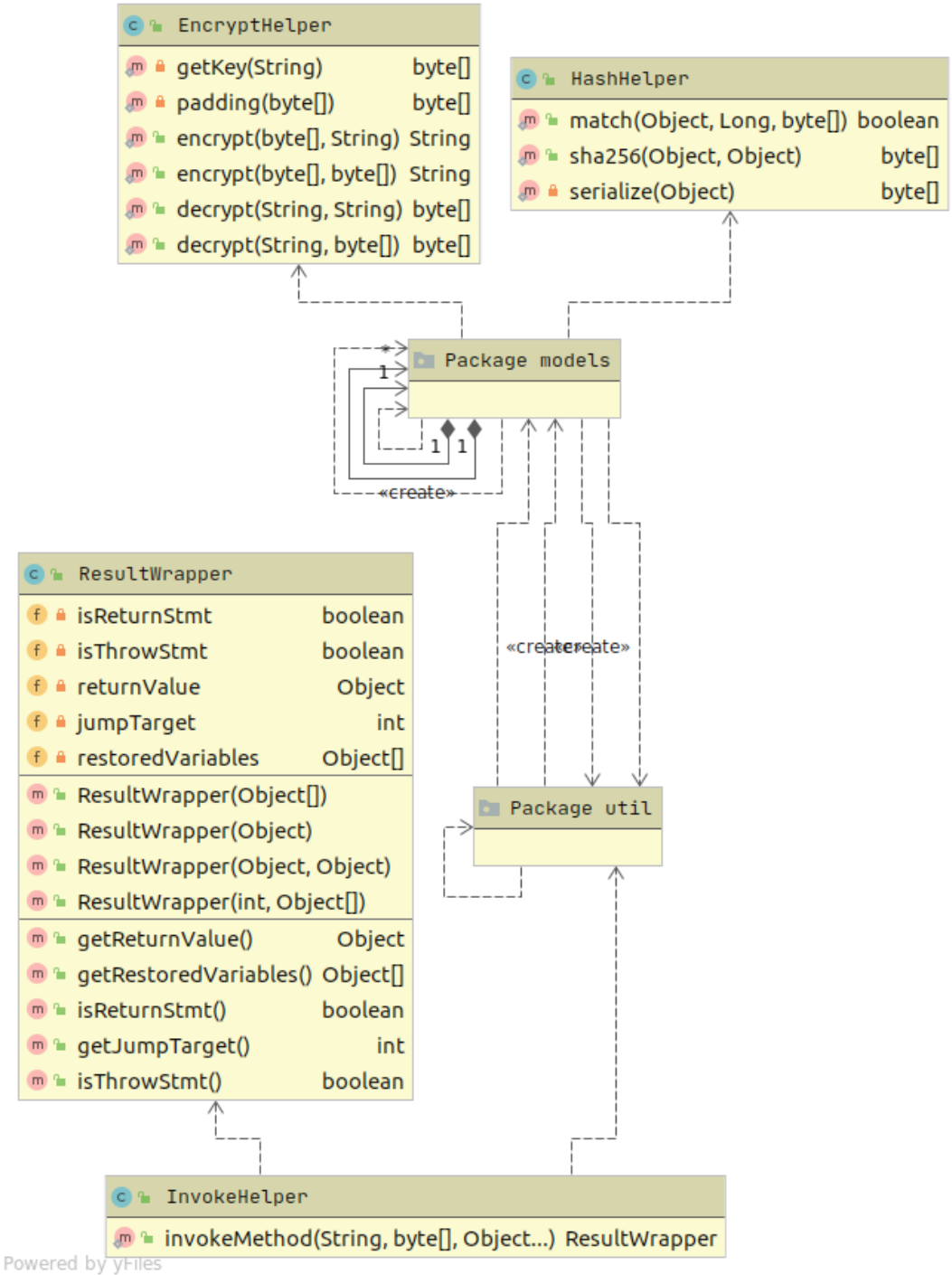


Figure A.8: Package embedded

References

[1] "Mobile operating system market share worldwide." https://gs.statcounter.com/os-market-share/mobile/worldwide/#monthly-201909-201909-bar, Accessed online: December 27, 2020. 1

[2] "Android developer guide." https://developer.android.com/guide, Accessed online: December 27, 2020. 1

[3] A. Desnos and G. Gueguen, "Android: From reversing to decompilation," *Proc. of Black Hat Abu Dhabi*, 01 2011. 1

[4] K. Tam, A. Feizollah, N. Anuar, R. Salleh, and L. Cavallaro, "The evolution of android malware and android analysis techniques," *ACM Computing Surveys*, vol. 49, pp. 1–41, 01 2017. 2

[5] X. Zhan, T. Zhang, and Y. Tang, "A comparative study of android repackaged apps detection techniques," *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 321–331, 2019. 2

[6] L. Li, T. Bissyandé, and J. Klein, "Rebooting research on detecting repackaged android apps: Literature review and benchmark," *IEEE Transactions on Software Engineering*, vol. PP, pp. 1–1, 02 2019. 2, 9

[7] "Antimine - minesweeper," Accessed online: December 27, 2020. 3, 46

[8] "Android app bundle," Accessed online: December 27, 2020. 5

[9] "backsmali," Accessed online: December 27, 2020. 6

[10] "App stores list (2019)," Accessed online: December 27, 2020. 6

[11] "Market share of the android app stores in china as of may 5, 2020," Accessed online: December 27, 2020. 6

[12] "Android: Publish your app." `https://developer.android.com/studio/publish`, Accessed online: December 27, 2020. 6

[13] "How can i verify the authenticity of an apk file i downloaded?." `https://android.stackexchange.com/questions/9312/how-can-i-verify-the-authenticity-of-an-apk-file-i-downloaded`, 2011. 6

[14] "Apktool," Accessed online: December 27, 2020. 8, 29

[15] J. Miecznikowski and L. Hendren, "Decompiling java bytecode: Problems, traps and pitfalls," pp. 111–127, 04 2002. 8

[16] "Frida," Accessed online: December 27, 2020. 8, 36

[17] S. Rastogi, K. Bhushan, and B. B. Gupta, "Android applications repackaging detection techniques for smartphone devices," *Procedia Computer Science*, vol. 78, pp. 26–32, 04 2016. 9

[18] K. Tian, D. Yao, B. Ryder, and G. Tan, "Analysis of code heterogeneity for high-precision classification of repackaged malware," pp. 262–271, 05 2016. 9

[19] Y.-D. Lin, Y.-C. Lai, C.-H. Chen, and H.-C. Tsai, "Identifying android malicious repackaged applications by thread-grained system call sequences," *Comput. Secur.*, vol. 39, pp. 340–350, 2013. 9

[20] H. Gonzalez, N. Stakhanova, and A. Ghorbani, "Droidkin: Lightweight detection of android apps similarity," vol. 152, 09 2014. 9

[21] W. Zhou, X. Zhang, and X. Jiang, "Appink: Watermarking android apps for repackaging deterrence," *ASIA CCS 2013 - Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, 05 2013. 9

[22] 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*, vol. 52, p. 102463, 2020. 11, 12

[23] S. Aonzo, G. C. Georgiu, L. Verderame, and A. Merlo, "Obfuscapk: An open-source black-box obfuscation tool for android apps," *SoftwareX*, vol. 11, p. 100403, 2020. 12, 63, 69

[24] "Proguard," Accessed online: December 27, 2020. 12

[25] 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*, CODASPY '14, (New York, NY, USA), p. 199–210, Association for Computing Machinery, 2014. 12

[26] 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*, vol. 7, 06 2019. 12

[27] G. Na, J. Lim, S. Lee, and J. Yi, "Mobile code anti-reversing scheme based on bytecode trapping in art," *Sensors*, vol. 19, p. 2625, 06 2019. 13

[28] B. Rimé and L. Schiaratura, "Gesture and speech in fundamentals of non-verbal behavior," 01 1991. 13

[29] 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*, pp. 129–138, 2015. 14

[30] "Obfuscator-llvm," Accessed online: December 27, 2020. 15

[31] "F-droid," Accessed online: December 27, 2020. 16, 69

[32] "Android developers. ui/application exerciser monkey," Accessed online: December 27, 2020. 16, 28, 73

[33] 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)*, pp. 550–561, 2016. 17, 31, 35, 36

[34] 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)*, pp. 25–32, 2017. 19, 30

[35] 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*, vol. 17, no. 8, pp. 1879–1893, 2018. 21, 39

[36] "Anzhi market," Accessed online: December 27, 2020. 23

[37] 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*, CGO 2018, (New York, NY, USA), p. 50–61, Association for Computing Machinery, 2018. 24, 31, 33, 35, 36, 41

[38] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, (New York, NY, USA), p. 224–234, Association for Computing Machinery, 2013. 26, 32

[39] S. Tanner, I. Vogels, and R. Wattenhofer, *Protecting Android Apps from Repackaging Using Native Code*, pp. 189–204. 04 2020. 26, 31, 37, 41, 68

[40] "Protecting android apps from repackaging using native code - repackaging protection," Accessed online: December 27, 2020. 26, 31, 44

[41] "Capec-167: White box reverse engineering," Accessed online: December 27, 2020. 29

[42] "dex2jar," Accessed online: December 27, 2020. 29

[43] R. Norboev, "On the robustness of stochastic stealthy network against android app repackaging *," 2018. 31, 32, 33, 36, 39, 41

[44] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood, "Testing android apps through symbolic execution," *SIGSOFT Softw. Eng. Notes*, vol. 37, p. 1–5, Nov. 2012. 32

[45] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," in *2007 IEEE Symposium on Security and Privacy (SP '07)*, pp. 231–245, 2007. 32

[46] "Mobile security testing guide," Accessed online: December 27, 2020. 33

[47] L. K. Yan and H. Yin, "Droidscope: Seamlessly reconstructing the OS and dalvik semantic views for dynamic android malware analysis," in *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, (Bellevue, WA), pp. 569–584, USENIX, 2012. 33

[48] S. Bhandari, W. B. Jaballah, V. Jain, V. Laxmi, A. Zemmari, M. S. Gaur, M. Mosbah, and M. Conti, "Android inter-app communication threats and detection techniques," *Computers & Security*, vol. 70, pp. 392 – 421, 2017. 33

[49] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang, "Fine-grained power modeling for smartphones using system call tracing," in *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, (New York, NY, USA), p. 153–168, Association for Computing Machinery, 2011. 33

[50] "fridump," Accessed online: December 27, 2020. 34

[51] "objection," Accessed online: December 27, 2020. 34

[52] "Radare - libre and portable reverse engineering framework," Accessed online: December 27, 2020. 34

[53] "r2frida," Accessed online: December 27, 2020. 34

[54] "Artdroid," Accessed online: December 27, 2020. 35

[55] "Xposed," Accessed online: December 27, 2020. 36

[56] "Owasp tampering and reverse engineering," Accessed online: December 27, 2020. 37

[57] "Capec-157: Sniffing attacks." https://capec.mitre.org/data/definitions/157.html, Accessed online: December 27, 2020. 38

[58] "Capec-609: Cellular traffic intercept." https://capec.mitre.org/data/definitions/609.html, Accessed online: December 27, 2020. 38

[59] "Owasp m3: Insecure communication." https://owasp.org/www-project-mobile-top-10/2016-risks/m3-insecure-communication, Accessed online: December 27, 2020. 38

[60] "Capec-94: Man in the middle attack." https://capec.mitre.org/data/definitions/94.html, Accessed online: December 27, 2020. 38

[61] "Protect against security threats with safetynet," Accessed online: December 27, 2020. 38

[62] "Jadx - dex to java decompiler," Accessed online: December 27, 2020. 50

[63] "Ghidra - software reverse engineering (sre) suite," Accessed online: December 27, 2020. 51

[64] "Android jni tips," Accessed online: December 27, 2020. 62

[65] "Soot - a java optimization framework," Accessed online: December 27, 2020. 64, 65

[66] "Developer android - extract native libs," Accessed online: December 27, 2020. 69

[67] "Apkpure," Accessed online: December 27, 2020. 69

[68] "Android developer," Accessed online: December 27, 2020. 73

[69] E. Coronado, J. Villalobos, B. Bruno, and F. Mastrogiovanni, "Gesture-based Robot Control: Design Challenges and Evaluation with Humans," 05 2017.

[70] "Owasp m9: Reverse engineering." https://owasp.org/www-project-mobile-top-10/2016-risks/m9-reverse-engineering, Accessed online: December 27, 2020.

[71] "Owasp m8: Code tampering." https://owasp.org/www-project-mobile-top-10/2016-risks/m8-code-tampering, Accessed online: December 27, 2020.

[72] "Code signing guide." https://developer.apple.com/library/archive/documentation/Security/Conceptual/CodeSigningGuide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40005929, Accessed online: December 27, 2020.

[73] A. Salem, F. F. Paulus, and A. Pretschner, "Repackman: A tool for automatic repackaging of android apps," in *Proceedings of the 1st International Workshop on Advances in Mobile App Analysis*, pp. 25–28, ACM, 2018.

[74] "Apk file extension." https://fileinfo.com/extension/apk, Accessed online: December 27, 2020.

[75] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *2012 IEEE Symposium on Security and Privacy*, pp. 95–109, 2012.

[76] G. Na, J. Lim, S. Lee, and J. Yi, "Mobile code anti-reversing scheme based on bytecode trapping in art," *Sensors*, vol. 19, p. 2625, 06 2019.

[77] C. Ren, K. Chen, and P. Liu, "Droidmarking: Resilient software watermarking for impeding android application repackaging," in *ASE 2014 - Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE 2014 - Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, pp. 635–645, Association for Computing Machinery, Inc, Jan. 2014.

[78] "Number of apps available in leading app stores as of 1st quarter 2020," Accessed online: December 27, 2020.

[79] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM – software protection for the masses," in *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015* (B. Wyseur, ed.), pp. 3–9, IEEE, 2015.

[80] S. Banescu and A. Pretschner, *A Tutorial on Software Obfuscation*. 01 2017.

[81] M. Miric, ""competition, innovation and signaling among app developers: A study of the jailbreak marketplace"," *Academy of Management Proceedings*, vol. 2013, no. 1, p. 13947, 2013.

[82] C. Mönch, G. Grimen, and R. Midtstraum, "Protecting online games against cheating," in *Proceedings of 5th ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '06, (New York, NY, USA), p. 20–es, Association for Computing Machinery, 2006.

[83] Y. Tian, E. Chen, X. Ma, S. Chen, X. Wang, and P. Tague, "Swords and shields: A study of mobile game hacks and existing defenses," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ACSAC '16, (New York, NY, USA), p. 386–397, Association for Computing Machinery, 2016.

[84] D. Bethea, R. A. Cochran, and M. K. Reiter, "Server-side verification of client behavior in online games," *ACM Trans. Inf. Syst. Secur.*, vol. 14, Dec. 2008.

[85] "Cordova framework," Accessed online: December 27, 2020.