Journal of Information Security and Applications 52 (2020) 102463

Contents lists available at ScienceDirect

Journal of Information Security and Applications

journal homepage: www.elsevier.com/locate/jisa

# A large-scale study on the adoption of anti-debugging and anti-tampering protections in android apps

Stefano Berlato[a], Mariano Ceccato[b,*]

[a] Fondazione Bruno Kessler, Trento, Italy
[b] Computer Science department, University of Verona, Italy

ARTICLE INFO

*Article history:*
Available online 28 February 2020

*Keywords:*
Anti-debugging
Anti-tampering
Android apps
Static analysis

ABSTRACT

Android apps are subject to malicious reverse engineering and code tampering for many reasons, like premium features unlocking and malware piggybacking. Scientific literature and practitioners proposed several Anti-Debugging and Anti-Tampering protections, readily implementable by app developers, to empower Android apps to react against malicious reverse engineering actively. However, the extent to which Android app developers deploy these protections is not known.

In this paper, we describe a large-scale study on Android apps to quantify the practical adoption of Anti-Debugging and Anti-Tampering protections. We analyzed 14,173 apps from 2015 and 23,610 apps from 2019 from the Google Play Store. Our analysis shows that 59% of these apps implement neither Anti-Debugging nor Anti-Tampering protections. Moreover, half of the remaining apps deploy only one protection, not exploiting the variety of available protections. We also observe that app developers prefer `Java` to `Native` protections by a ratio of 99 to 1. Finally, we note that apps in 2019 employ more protections against reverse engineering than apps in 2015.

## 1. Introduction

Being the most diffused operating system for smartphones, Android presents a way for developers to share their apps with billion end-users. Moreover, many of these apps produce revenues through advertisements, in-app purchases, direct sales or subscriptions to premium features. In these cases, apps embed valuable assets that their developers want to protect. The possibility to steal such assets attracted several malicious attackers. Unfortunately, attackers can easily recover source code from compiled Android apps. Then, attackers can tamper with the logic of the apps to their advantage, repackage them and distribute them again. In the last six years, the scientific community published more than 57 research papers [1] on repackaged apps, highlighting how the problem is relevant and actual. Spotify is a perfect example of how this can happen. Many attackers studied how to tamper with the code of Spotify to unlock premium features for free. Then, they published the tampered versions of Spotify on the internet, available for everyone to download. In the end, Spotify had so many tampered versions available on the internet that the developers had to take drastic countermeasures. The developers cracked down and banned several accounts who they thought were using tampered versions of Spotify [2]. Another remarkable example is the paid mobile game “Monument Valley”. The owner company reported that just 5% of the end-users paid for downloading the game from the Google Play Store [3]. All the other end-users obtained a tampered version from third-party app stores or other sources.

Ceccato et al. [4] studied in detail the behaviours and strategies adopted by attackers performing malicious reverse engineering. They found that dynamic analysis through debugging is a prominent step for both identifying the portion of the code to attack and for validating the results of the attack. Therefore, it seems that debugging and tampering are the two most effective strategies to attack and Android app.

Android apps developers can leverage many protections to mitigate or delay a tampering attack. Anti-Debugging (AD) and Anti-Tampering (AT) are two categories of protections that mitigate these attack strategies. Differently from passive Obfuscation techniques where the code of the app is changed to make it harder to understand, AD and AT protections allow an app to react against malicious reverse engineering actively at run-time. In particular, AD protections give the app the ability to (i) prevent a debugger to attach to the process of the app; (ii) spot the presence of a debugger or an emulated environment at run time; (iii) tamper with the data structures of the debugger to hinder its correct functioning. AT protections allow the app to (i) detect alterations from its

* Corresponding author.
*E-mail addresses:* sberlato@fbk.eu (S. Berlato), mariano.ceccato@univr.it (M. Ceccato).

https://doi.org/10.1016/j.jisa.2020.102463
2214-2126/© 2020 Elsevier Ltd. All rights reserved.

original state by checking the integrity of the code; (ii) verify the source of the app itself (i.e. the app store where the app comes from). App developers can find many suggestions on how to implement these protections both in the literature [5–7] and in other informal resources, like the official Android Studio documentation [8] and the OWASP Mobile Security Testing Guide [9].

However, there is no systematic study that quantifies how often app developers employ these protections. We present a large-scale study conducted to shed light on the adoption of AD and AT protections in Android apps. To the best of our knowledge, this is the first work to assess the frequency of usage of such protections. We analyzed 14,173 apps from 2015 and 23,610 apps from 2019 from the top apps in the Google Play Store. The results are quite surprising: only 41% of these apps actively implement at least one AD or AT protection. Moreover, half of this 41% deploy only one protection, not exploiting the variety of available protections. App developers prefer to deploy simpler `Java` protections than `Native` ones with a ratio of 99 to 1. Unfortunately, `Java` protections are also easier to bypass, since attackers can easily recover the source code. Moreover, we observe that apps from 2019 employ more protections against reverse engineering than apps from 2015.

The paper is structured as follows. In Section 2, we present a survey on AD and AT protections. It is a catalogue of protections along with a brief high-level description and an example implementation. In Section 3, we describe our approach to classify the main programming elements of each protection and how they compose into a unique protection fingerprint. We use these fingerprints to detect protections in the code of apps. We also describe our tool, called `ATADetector`, for automating protection detection. Afterwards, in Section 4, we incrementally refine the fingerprints to improve detection accuracy. In Section 5, we define the research questions and we present the large-scale study we conducted to answer them. In Section 6, we discuss technical limitations and threats to validity. Eventually, after a discussion on related work in Section 7 and future work in Section 8, we conclude the paper in Section 9.

## 2. Survey of anti-debugging and anti-tampering protections

This section presents our categorization of AD and AT protections. First, we briefly describe the attack model assumed by these protections. Then, we describe our approach for performing the survey. Finally, we present and discuss each identified protection.

### 2.1. Attack model

Following the results described by Ceccato et al. [4], we consider a malicious reserve engineering activity, in which one or more attackers aim at altering the functioning of an app to gain some advantage. The first step is code comprehension. The attackers have to unveil the logic behind the app by investigating its code. Consequently, the attackers can understand where and how to modify the app to achieve their specific goals. The most prominent technique attackers use is dynamic analysis through debugging [4]. This process usually consists of attaching a debugger to the process of the app. Using the debugger, the attackers can monitor the status of the app and even control its execution flow. By controlling the instructions to execute next, the attackers can gain deep insights on the functioning of the app. Finally, the attackers can change the code of the app. This last operation is commonly known as *Tampering*. The attackers tamper with one or more portions of the code of the app to modify its functioning toward specific outcomes. For instance, suppose an app with premium features. The attackers could tamper with the portion of the code that checks whether the premium subscription is expired or not to always enjoy premium features. Therefore, we consider two categories of protections against malicious reverse engineering: Anti-Debugging and Anti-Tampering.

### 2.2. AD and AT protections survey

To gather AD and AT protections, we start from the resource Android app developers consult more often, thus the Internet. Balebako et al. [10] studied the behaviour of app developers about privacy and security. One of their findings is that app developers "*simply searched online when they were looking for advice*". Also, Balebako et al. found that developers navigate websites like *Hackernews, TreeHouse* and *StackExchange* for security-related researches. Therefore, we analyze this informal literature to identify descriptions of AD and AT protections. Also, we analyze the Android official documentation [8], OWASP security guidelines [9], security blogs Alexander-Bown and code repositories [12,13]. This survey allows us to define 5 AD protections and 4 AT protections.

### 2.3. Anti-Debugging protections

- *Emulator detection*: Attackers may take advantage of Android emulators to monitor the status of an app. Attackers can read the values of program variables and sniff Internet traffic, inferring valuable information about the functioning of the app. However, Android emulators have several default configuration values that app developers can detect. Therefore, app developers can insert in their apps mechanisms to inspect system properties to check whether the app executes in an emulator. For instance, it is common to read the model or the manufacturer of the smartphone to compare it against values related to Android emulators, like "generic" or "goldfish".
- *Dynamic analysis framework detection*: Similar to Android emulators, dynamic analysis frameworks allow attackers to gain insights on the functioning of an app. These frameworks, like Taintdroid [14], Xposed[1] and Frida,[2] run on real Android devices and allow manipulating the runtime environment by hooking API calls to return spurious values. For instance, whenever the app is requesting its digital signature through the *PackageInfo.signingInfo* attribute, the attackers could use Xposed to intercept this invocation and return whatever value they like. Moreover, these frameworks allow monitoring the status of Android apps and dynamically altering their behaviour. Detecting these runtime modifications is not easy. Therefore, the focus of the protection is often on spotting the presence of these frameworks in the smartphone. The simplest way is to scan package names, files or binaries to look for resources known to be components of these frameworks
- *Debugger detection*: Android supports two debugging protocols: `Java` level through the *Java Debug Wire Protocol* (`JDWP`) and Linux level with *GNU Debugger* (`GDB`). Usually, developers employ debuggers in the testing phase for findings bugs in their apps. However, attackers can use debuggers to send commands to the app and alter the execution flow or the values within program variables. For instance, an attacker could tamper with the value of the variable holding the amount of virtual money in a game app. To be fully protected, an app has to implement protections against both levels of debugging. An app can detect a `JDWP` debugger by invoking the available API through both `Java` and `Native`

[1] https://www.xda-developers.com/xposed-framework-hub/.
[2] https://www.frida.re/.

code, and the GDB debugger by checking whether an extra process (i.e. the GDB debugger) is attached to the process of the app or not. Beside debugger detection strategies, there are also preventive strategies. For example, only one process at a time can work as a debugger of another process. Therefore, an app can attach to itself a mock debugger process to prevent a real GDB debugger process from attaching, because the attachment interface is already engaged.
- *Debuggable status detection*: To make an app available for debugging in an Android device, the attackers have to alter the "debuggable" flag in the manifest file. This way, Android will start an extra thread for handling the JDWP protocol. Checking the value of this flag gives a clear indication of the debuggable status of the app.
- *Altering debugging memory structure*: The status of the global virtual machine in which an app is running is accessible through the `DvmGlobals` structure that contains several variables crucial for the functioning of the JDWP debugger. In *Dalvik*, the Android virtual machine until Android version 5.0 (Lollipop), there is the global variable `gDvm` that points to this structure. In *ART*, the new Android RunTime system from Android version 5.0, this variable is not available anymore. However, the *ART* runtime exports some pointers related to JDWP as global symbols. Therefore, in both cases, an app can manipulate the behaviour of the debugger by overwriting these variables. For instance, an app could replace the address of the function `jdwpAdbState::ProcessIncoming` with the address of the function `JdwpAdbState::Shutdown` [9]. This will cause the debugger to disconnect immediately.

### 2.4. Anti-tampering protections

- *Signature Checking*: Tampering an app usually implies the modification of its code. Then, the attackers have to repackage the new version of the code into an Android PacKage (*APK*) file that end-users will install on their smartphones. Since the Android operating system requires *APK*s to have a digital signature to check upon installation, the attackers need to sign the *APK* file again. The attackers cannot access the private key of the original developers. So, the attackers will sign the *APK* file with a different key. Therefore, the most trivial protection against tampering is to compare the current signature of the *APK* file with the original one. The app can obtain the current signature through dedicated APIs using the `PackageManager.GET_SIGNATURES` and the `PackageInfo.signatures` (until Android version 8.0) or `PackageManager.GET_SIGNING_CERTIFICATES` and `PackageInfo.signingInfo` (from Android version 9.0) APIs.
- *Code integrity checking*: Following the same concept of the previous protection, Code Integrity Checking is another similar protection. This time, the app computes a digest value on a specific resource or file and then compare it with the expected value. Therefore, an app can access and hash the file containing the `Java` code (i.e. the *.dex* file) and check whether this value is equal to the expected value or not. App developers can use standard libraries like `Zipentry`[3] to automatically obtain useful values like the Cyclic Redundancy Check (CRC) error-detecting code.
- *Installer verification*: To avoid detection, usually, attackers publish tampered and repackaged apps in third-party app stores [15]. When installing an app, the Android operating system keeps track of the app store where the APK file comes from. The app can invoke the `PackageManager.getInstallerPackageName` API that returns the package name of the app through which the end-user installed the current app. The protection consists of checking whether this value is consistent with the app stores where the developers published their app. Let's suppose the developers published their app only in the Google Play Store. End-users should have installed the app through the Play Store app that has "*com.android.vending*" as the package name. If the value returned by the `PackageManager.getInstallerPackageName` API is "cm.aptoide.pt", the app was installed from Aptoide,[4] an independent Android app store. Therefore, some attackers likely tampered the app and published it on Aptoide.
- *SafetyNet attestation*: `SafetyNet` [16] is a platform security service offered by Google [16]. An app can invoke `SafetyNet` to verify the integrity of the smartphone in which it is running. However, `SafetyNet` can also provide information about the app that invoked the service, like the signature. Therefore, this information can be used to perform integrity checks on the app itself.

### 2.5. Exclusions

Developers can implement many other protections in their apps, that we decided to exclude:

- *Root detection*: A end-user can obtain superuser permissions over an Android smartphone through a process called *"Rooting"*. With superuser permissions, it is possible to alter system settings, access private areas in the primary memory and install specialized apps. For instance, with superuser permissions, an attacker can install dynamic analysis frameworks like Xposed. Even though providing significant insights about the smartphone where the app runs, this protection does not address AD or AT directly. Indeed, this protection provides information about the status of the smartphone rather than on the app itself.
- *File storage integrity checking*: Some apps may externally download code and resources after they are installed and then perform checks on them, but this is a discouraged feature [17]. Therefore, an app would not implement this protection not because the developers are overlooking security, but because downloading code after installation is a feature not implemented in the app.
- *Time-checks*: Another way to detect debuggers is to implement time-checks. The possibility to insert breakpoints in the code is one of the most useful features of a debugger. This allows analyzing the execution flow of the app and the status of the variables. However, this also halts the execution of the process. Therefore, an app can monitor the elapsed time between two instructions. If this time is longer than a pre-defined threshold, a debugger has most probably halted the execution in between the operations with a breakpoint. However, an app may query for the time for many reasons, like performance evaluation, alerts or scheduled notifications. Therefore, this protection is problematic to detected and it would suffer many false positives.

## 3. Definition of protection fingerprints

We now present our method for the detection of the protections in Android apps. In this section, we describe the general

[3] https://developer.android.com/reference/java/util/zip/ZipEntry.

[4] https://www.aptoide.com/en/home.

**Table 1**
Set of protection atoms for the *installer verification protection at java level*.

| | | |
|---|---|---|
| Classes | c1 | android/content/Context |
| | c2 | android/content/pm/PackageManager |
| Methods | m1 | android/content/Context.getPackageName |
| | m2 | android/content/Context.getPackageManager |
| | m3 | android/content/pm/PackageManager.getInstallerPackageName |
| Attributes | | |
| Strings | s1 | *com.android.vending* |

```
1   private static final String PLAY_STORE_APP_ID = "com.android.vending";
2
3
4   public static boolean verifyInstaller(final Context context) {
5
6     final String installer = context.getPackageManager()
7       .getInstallerPackageName(context.getPackageName());
8
9     return installer != null
10      && installer.startsWith(PLAY_STORE_APP_ID);
11  }
```

**Fig. 1.** Example Implementation of *Installer Verification* Protection.

approach with a concrete example for one protection. Then we illustrate how we combine the elements of each protection to create a fingerprint. Finally, we present the tool we developed for the automatization of the protection detection.

### 3.1. General approach for protections atoms identification

Starting from the description of each protection, we analyze instruction-by-instruction which are the most characterizing programming elements. From each instruction, we extract the essential elements in terms of classes, methods, attributes (`Java`), imported symbols (`C++`) and strings (`Java` and `C++`) used in the code. The result is a collection of programming elements that together identify the protection. When found in the code, these elements are clues that the developers deployed the protection in their app. We call these elements “protection atoms”. We applied this approach for every protection for both `Java` and `C++` implementations. For instance, Fig. 1 (page 5) shows the implementation for the *Installer Verification* protection proposed by Alexander-Bown [11].

The snippet of code in Fig. 1 (page 5) checks whether the end-user installed the app from the Google Play Store. To achieve this objective, the code declares a string variable containing the package name of the Google Play Store app, that is “*com.android.vending*” (line 1). Then it defines a function `verifyInstaller` (line 5). Given an instance of the `Context` object, this function gets the package name of the installer (lines 6-7). The function tests whether the string is empty or not (line 9). If the end-user installed the app from an *APK* file manually and not from an app store, this could happen. Finally, the function checks whether the string is equal to the package name of the Google Play Store app (line 10). If this is the case, the end-user installed the app through the Google Play Store. Otherwise, the app comes from another source. In case the developers originally published their app only in the Google Play Store, this is an indication of possible tampering attempt. It implies that someone else downloaded the app, most probably modified it, and then published it in another app store.

From this snippet of code, we extract the relevant protection atoms that allow us to conjecture the presence of the *Installer Verification* protection. Table 1 (page 6) reports these protection atoms.

We include the two `Java` classes employed in the snippet of code, `Context` and `PackageManager`. Then, we add the methods related to these classes that are involved in the implementation of the protection, that are `Context.getPackageName`, `Context.getPackageManager` and `PackageManager.getInstallerPackageName`. For instance, the method `m1` (`Context.getPackageName`) returns the package name of the current app. The method `m2` (`Context.getPackageManager`) returns an instance of the class `c2` (`PackageManager`), while the third returns the package name of the app that installed the current app. In our case, this package name is expected to be equal to the string `s1` (“*com.android.vending*”).

The baseline assumption is that, if an app contains these protection atoms, it likely implements the *Installer Verification* protection. A similar argument applies to the other protections as well, both at the `Java` and at the `Native` level. We listed all the protection atoms in Appendix B.

### 3.2. Boolean formula applied on protection atoms

We introduce a boolean formula applied over the protection atoms to connect them through `AND` and `OR` operators. This formula describes which protection atoms we have to detect to reasonably suppose that the app is implementing the related protection. We define as “fingerprint” the combination of the protection atoms with a boolean formula. To identify the *Installer Verification* protection in an app based on the protection atoms in Table 1 (page 6), we need to detect both the method `m3` (`PackageManager.getInstallerPackageName`) and the string `s1` (“*com.android.vending*”). These are the essential protection atoms without which it is very difficult to implement this protection. The returning value of the method and the string have to be compared to determine whether the end-user installed the app from the Google Play Store or not. Therefore, the fingerprint for the *Installer Verification* protection at `Java` level is:

`m3 AND s1`

We report the fingerprints of other protections in Appendix C.

**Table 2**
Extended set of protection atoms for the *Installer Verification* protection at Java level.

| | | |
|---|---|---|
| Classes | c1 | android/content/Context |
| | c2 | android/content/pm/PackageManager |
| Methods | m1 | android/content/Context.getPackageName |
| | m2 | android/content/Context.getPackageManager |
| | m3 | android/content/pm/PackageManager.getInstallerPackageName |
| Attributes | | |
| Strings | s1 | *com.android.vending* |
| | s2 | *android.content.pm.PackageManager* |
| | s3 | *getInstallerPackageName* |

```java
private static final String PLAY_STORE_APP_ID = "com.android.vending";
private static final String className = "android.content.pm.PackageManager";
private static final String methodName = "getInstallerPackageName";


public static boolean verifyInstaller(final Context context) {

  Class<?> packageManagerClass = Class.forName(className);

  Method installerMethod =
    packageManagerClass.getMethod(methodName, String.class);

  final String installer = installerMethod.invoke(
    context.getPackageManager(), context.getPackageName());

  return installer != null
    && installer.startsWith(PLAY_STORE_APP_ID);
}
```

**Fig. 2.** Example Implementation of *Installer Verification* Protection with Reflection.

## 3.3. Handling reflection

The developers could have hidden some protection atoms through Java Reflection to harden their protections against attackers. Reflection is a peculiar feature in Java that allows an executing program to access variables and methods dynamically by name. For instance, developers can replace a direct invocation to a method with a reflective call. Fig. 2 (page 5) shows the implementation for the *Installer Verification* protection with a reflective invocation to the `getInstallerPackageName` method.

The *Installer Verification* protection implemented in the snippet of code in Fig. 2 (page 7) is as effective as the original one implemented in Fig. 1 (page 5). However, the `PackageManager.getInstallerPackageName` method is invoked through Reflection. The code declares two variables containing the class (line 2) and the method (line 3) as strings. They are "*android.content.pm.PackageManager*" and "*getInstallerPackageName*", respectively. Then, the code obtains the method (lines 10-11) and invokes it (lines 13-14). Therefore, the code includes two strings containing (i) the fully-qualified name (FQN) of the class and (ii) the method to invoke. To make our approach more effective, we can include such strings in our protection atoms. Note that an app can have a hybrid approach, accessing the class traditionally and the method through reflection. In this case, we would search for the class as a symbol and the method as a string.

A more systematic approach to solve reflective calls in Java is proposed by Li et al. [18]. Their approach is based on constant propagation with static analysis, to compute the strings used as class and method names in reflective calls, and the strings used as class and field names in reflective field accesses. However, considering that their approach would be expensive, but deliver partial results when strings are obfuscated or encrypted, we opted for a faster and cheaper alternative.

We report in Table 2 (page 7) the extended set of protection atoms.

In Table 2 (page 7), we include the strings necessary to invoke the method `m3` through reflection (strings `s2` and `s3`). Therefore, the final fingerprint for this protection at Java level is:
`(m3 OR (s2 AND (c2 OR s3))) AND (s1)`

The first half of the fingerprint refers to the retrieving of the package name of the installer app. The app can obtain this package name either directly (`m3`) or through reflection, with the name of the method as a string (`s2`) and the class, either importing it (`c2`) or getting it through reflection as well (`s3`). The second half of the fingerprint refers to the detection of the "*com.android.vending*" string. From now on, to avoid complications in the fingerprints, we omit this mechanism for Reflection detection in the fingerprints.

## 3.4. Concerns on Fingerprint Fragility

We observed that expecting to detect *all* the protection atoms of the protections may be an ineffective and a too strict requirement. Therefore, there are two concerns to discuss:

- We have to exclude the protection atoms that a developer can use for other reasons besides implementing AD and AT protections. For instance, an app can use the `Context` class also for checking available permissions or creating a new object, like an `android/view/View` object. Moreover, an app can use the "*com.android.vending*" string for in-app purchases. So, the signature should be flexible and exclude those protection atoms that might occur spuriously also outside of protection code.

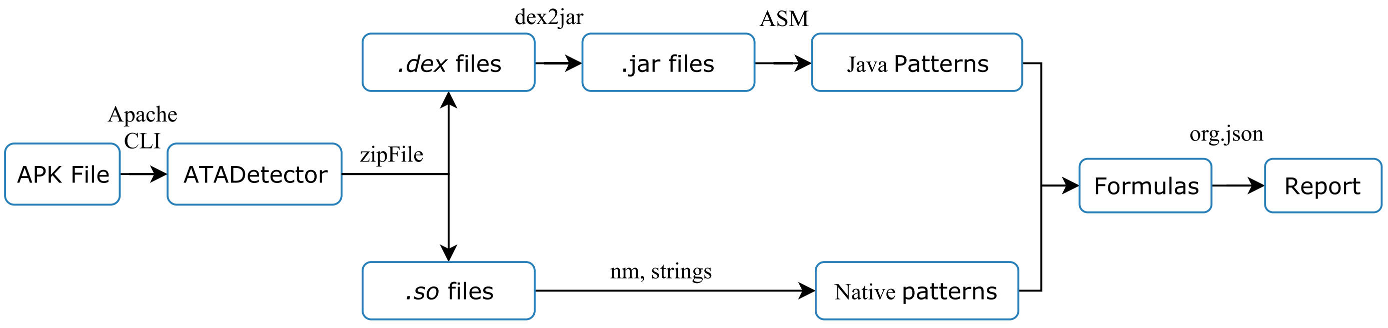


**Fig. 3.** `ATADetector` workflow.

- A developer could have deployed the protection in a slightly different way by referencing to alternative implementations. For example, the developers can obtain the package name of the app also through the attribute `PackageInfo.packageName` or by directly embedding the value as a string variable. Moreover, there are other app stores besides Google app store, like Samsung (*"com.sec.android.app.samsungapps"*) and Amazon (*"com.amazon.venezia"*) app stores. Thus, we need to extend the protection atoms to achieve more comprehensive and effective detection results to reduce the risk of overlooking protection implementations.

We need to refine our approach to achieve better detection results. Therefore, to face this challenge and define more accurate fingerprints, we have to test our fingerprints in an iterative process of incremental validation and refinement.

### *3.5. Tool implementation*

We automated the detection of the fingerprints in a tool named `ATADetector` (Anti-Tampering and Anti-Debugging Detector). Fig. 3 (page 9) summarizes the workflow of `ATADetector`. We employ the *Apache Commons CLI library*[5] to parse input arguments. `ATADetector` takes as input an *APK* file and splits the app in the `Java` (*.dex* files) and `C++` (*.so* files) components. `ATADetector` transforms the *.dex* files into *.jar* with the nightly version (2.1) of *dex2jar*.[6] We implemented the tool in `Java` on top of the *ASM* library [19]. This library allows parsing `Java` bytecode of an Android app to extract the programming elements like classes, methods, attributes and strings.

In Java, strings are immutable and stored in the *constant pool* of the class where they are used. We detect string values used in the Java bytecode by identifying their usages, i.e. the LDC (LoaD Constant) Java opcode. This opcode is meant to take a specific constant value from the constant pool and push it to the operand stack to be used by the subsequent opcode, for instance, to make a reflective call. The string value is provided by ASM, that resolves the argument of the LDC opcode.

At `Native` level, we extract imported symbols and strings with the Linux command-line utilities `nm`[7] and `strings`,[8] respectively. Finally, we combine the extracted protection atoms in the fingerprints and produce a JSON report with the *org.json*[9] library.

[5] https://commons.apache.org/.
[6] https://github.com/pxb1988/dex2jar/releases.
[7] https://linux.die.net/man/1/nm.
[8] https://linux.die.net/man/1/strings.
[9] https://github.com/stleary/JSON-java.

## 4. Incremental validation and refinement of protection fingerprints

Before using the fingerprints on the large case study, we carried out an iterative process for refining and then validating the fingerprints. The goal is to tune and adapt the fingerprints with more and more complex and complete experimental settings. In this section, we illustrate this process by presenting the three validations steps we performed.

### *4.1. Validation and refinement with toy apps*

For the first validation, we consider simple "Hello World" apps with no specific functionality. We manually deploy the protections one by one in the apps, following the example implementations illustrated in Section 2. We run `ATADetector` on these apps to verify whether it detects the protections in this most straightforward setting. To have cleaner results, we perform the analysis both on the whole code of the app and then only on the `Java` class that implements the protection. In this way, we can understand which protection atoms are significant for the detection of the protections. There can be protection atoms often used in protections but also for other purposes and in Android standard libraries as well. Thus, their detection would be an irrelevant contribution to the final result. In the analysis considering the `Java` class that implements the protection only, we identify all the protections correctly. In the analysis considering the whole app, we identify all the protections but also some false positives. Table 3 (page 9) summarizes the results of the analysis on the whole app in terms of true and false positives.

We have a false positive for the *DebuggableStatusDetection* protection because of the `ApplicationInfo.flags` attribute. We can suppose that Android standard libraries use this attribute and therefore we can remove it from our fingerprint. The same reasoning applies for the `PackageInfo.signatures` attribute found in the `FontsContractCompat` `Java` class, that results in a false positive for the *Signature Checking* protection. At `Native` level, we detect the `pthread_create` symbol in the "`string`" library and not only in the *Debugger Detection* protection, causing another false positive. Given these results, we refined the fingerprints in protection fingerprints by removing the protection atoms that are used not just by protections but also by other code.

Finally, this analysis allows us to check the accuracy of ATADetector and the ASM module. Indeed, ATADetector is able to identify every Java class, method, attribute and string value we insert in the toy apps for implementing the protections. However, we still have to investigate more complex settings with more complex apps.

**Table 3**
First validation on 10 “Hello World” apps.

| Category | Protection | True Positive | False Positive |
|---|---|---|---|
| AD | *Emulator Detection* | 1 | 0 |
| | *Dynamic Analysis Framework Detection* | 1 | 0 |
| | *Debugger Detection* | 1 | 1 |
| | *Debuggable Status Detection* | 1 | 1 |
| | *Altering Debugging Memory Structure* | 1 | 0 |
| AT | *Signature Checking* | 1 | 1 |
| | *Code Integrity Checking* | 1 | 0 |
| | *Installer Verification* | 1 | 0 |
| | *SafetyNet Attestation* | 1 | 0 |

**Table 4**
Second validation on 115 F-Droid APKs.

| Category | Protection | True Positive | False Positive |
|---|---|---|---|
| AD | *Emulator Detection* | 3 | 1 |
| | *Dynamic Analysis Framework Detection* | 0 | 2 |
| | *Debugger Detection* | 9 | 0 |
| | *Debuggable Status Detection* | 0 | 0 |
| | *Altering Debugging Memory Structure* | 0 | 0 |
| AT | *Signature Checking* | 6 | 0 |
| | *Code Integrity Checking* | 0 | 0 |
| | *Installer Verification* | 1 | 0 |
| | *SafetyNet Attestation* | 0 | 0 |

**Table 5**
Third validation on 50 google play store APKs.

| Category | Protection | True Positive | False Positive |
|---|---|---|---|
| AD | *Emulator Detection* | 13 | 1 |
| | *Dynamic Analysis Framework Detection* | 0 | 2 |
| | *Debugger Detection* | 11 | 4 |
| | *Debuggable Status Detection* | 0 | 0 |
| | *Altering Debugging Memory Structure* | 0 | 0 |
| AT | *Signature Checking* | 12 | 2 |
| | *Code Integrity Checking* | 1 | 0 |
| | *Installer Verification* | 11 | 0 |
| | *SafetyNet Attestation* | 2 | 0 |

## 4.2. Validation and refinement with open source apps

To further validate protection fingerprints, we analyze a batch of 115 apps downloaded from F-Droid,[10] an online repository that collects code of free and open-source apps. The availability of source code allows us to validate the results of `ATADetector` manually and distinguish true from false positives more easily. We choose these 115 apps by selecting the apps most downloaded from F-Droid. Table 4 (page 10) summarizes the results of the analysis in terms of true and false positives.

Being the apps open-source, we expect to detect only a few protections. In fact, `ATADetector` identifies only 22 protections in 115 apps. Then, we check whether each of these 22 protections is a true positive or a false positive. During this process, we observe that many protections come from third-party libraries, like *org.sufficientlysecure.donations*. Based on this observation, we collect the package names of these libraries to be able to filter them later.

Out of 22 cases, we identify 3 false positives only. One refers to the *Emulator Detection* protection. In this case, `ATADetector` identifies the presence of the string “nox”, the name of an Android emulator, and the `Build.DEVICE` attribute. An app can compare these two values to check whether it is running on an emulator. Unfortunately, the app uses the “nox” string elsewhere, so it is not part of the protection. However, the app implements the *Emulator Detection* protection by comparing the value of the `Build.DEVICE` attribute with the “generic” string. This string is often present in the properties of Android emulators. Unfortunately, we cannot consider it a peculiar protection atom because too commonly used. Therefore, the app implements the *Emulator Detection* protection, but `ATADetector` does not match it properly.

The other two false positives concern the *Dynamic Analysis Framework Detection*. Both of them are due to the detection of the “xposed” string. The first false positive is because the app was an Xposed module itself. The second is because the app inserted that string in an ad-blocker list.

[10] https://f-droid.org/.

## 4.3. Validation and refinement with closed source apps

We conduct a third validation against 50 apps randomly sampled from the Google Play Store. The source code of these apps is not accessible. Therefore, we validate the results of `ATADetector` by manually analyzing the code of the apps through reverse engineering. Table 5 (page 11) summarizes the results of this third validation.

We identify 60 protections, way more than in the previous validation. For each of them, we check whether it is a true positive or a false positive. As a result, we find that 10 of the 60 protections are false positives. As in the previous step of validation, we manage to separate between libraries and app code, relying on the package names. Even though obfuscated, we empirically notice that it is highly likely that an app retains the structure and the names of the packages.

The *Signature Checking* protection has two false positives because of the `provider.FontsContractCompat` class in Android standard libraries. This class contains the `PackageInfo.signatures` attribute, an essential protection atom for the detection of the protection that we cannot eliminate from the fingerprint. Therefore, we add peculiar strings found in the `provider.FontsContractCompat` class like "No package found for authority: ", "Found content provider " and ", but package was not" to the protection atoms. The idea is to use them to recognize this false positive. In practice, if `ATADetector` identifies all of these three strings, it will skip one occurrence of the `PackageInfo.signatures` attribute. The *Emulator Detection* protection has one false positive because of the detection of strings related to properties of Android emulators but too commonly used in Android app. We are referring to strings like "unknown", "Andy" and "vbox". Therefore, we removed them from the fingerprint.

## 5. Large-scale analysis

This section reports the process we followed for performing a large-scale analysis on Android apps along with the final results and considerations. We first formulate five research questions to guide the definition of our experimental settings. Then, we describe the datasets we analyzed and a set of metrics over the data. After an overview of the procedure we followed during the analysis, we conclude the section by answering each of the research questions.

### 5.1. Research questions

We formulate five research questions to guide our large-scale study:

1. **RQ$_1$**: How frequently do apps use AD and AT protections?
2. **RQ$_2$**: How frequently do protections integrate each other?
3. **RQ$_3$**: How frequently are AD and AT protections deployed in developers' code and in third-party libraries?
4. **RQ$_4$**: How frequently are AD and AT protections implemented at `Java` and at `Native` level?
5. **RQ$_5$**: What is the evolution in the adoption of AD and AT protections in apps?

The first research question aims at measuring the extent to which Android apps employ AD and AT protections.

The second research question relates to how many different protections an app implements and how they supplement each other. In particular, we want to investigate how they integrate when considering pairs of protections. This indicates how developers combine protections in their apps and what are the most popular pairings.

The third research question aims at distinguishing between protections implemented by the developers and the ones derived from third-party libraries, measuring the extent to which developers actually protect their apps.

AD and AT protections can be deployed both at `Java` and `Native` level. While it is easier to implement `Java` protections, `Native` ones are more difficult to bypass by attackers. The fourth research question aims to discover how frequently developers opt for one or the other.

The last research question assesses the evolution in the usage of AD and AT protections across years.

### 5.2. Metrics

To answer the research questions, we define the following metrics to apply on data resulting from the large-scale analysis:

- *Category* - Each app belongs to one or more categories of the Google Play Store (e.g., Education, Sport, Communication). Each category hints to the purpose of the app and the assets the developers want to protect. It is reasonable to suppose that the need to protect apps changes from one category to the other. We use this metric in RQ1.
- *Scope* - Android apps integrate many libraries developed by third-parties. We noticed that the developers of these libraries deploy AD and AT protections too. This metric, used in RQ3, specifies whether the protections derive from a third-party library or not.
- *Level* - Protections can be implemented both at `Java` and `Native` level, each having its advantages and drawbacks. For example, the deployment of `Native` protections requires more effort but leads to more effective protections [9]. This metric, employed in RQ4, allows identifying the programming language used for the implementation.
- *Year* - To give perspective to our analysis, we also consider top-category Android apps available in 2015. In this way, it is possible to track the evolution in the adoption of AD and AT protections in the last four years. We use this metric in RQ5.

### 5.3. Subjects apps

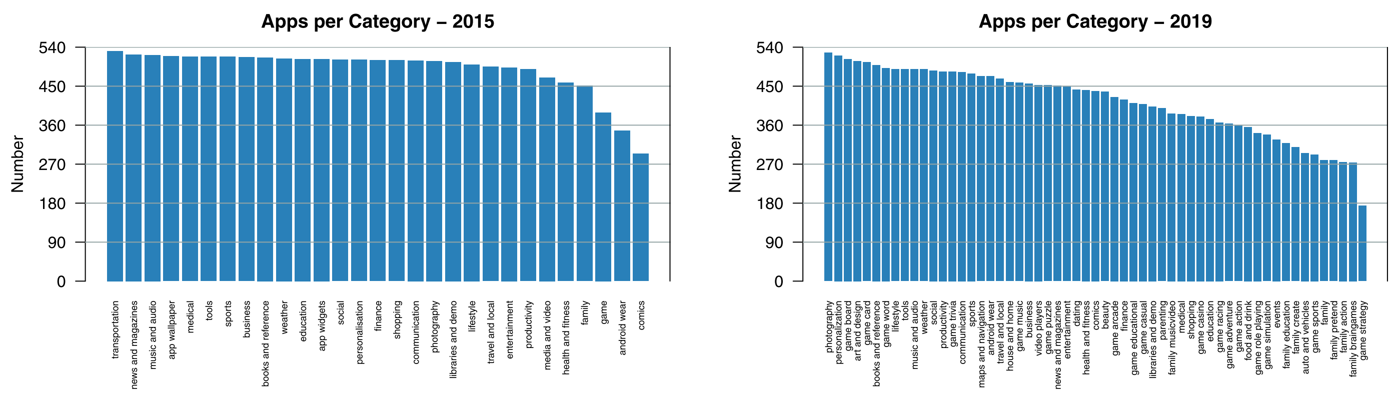
For our large-scale analysis, we employed two different datasets of top-category Android apps. We built both of them following the same process in 2015 and 2019. First, we crawled the Google Play Store to collect the package names of the top Android apps for each category. The Google Play Store limits the number of top apps for each category to 540. There were 29 categories in 2015 and 57 categories in 2019. Then, we searched for these apps in Androzoo [20], a collection of Android apps, and we downloaded those that were available. In the end, our datasets consist of 14,173 apps from 2015 and 23,610 from 2019. Fig. 4 (page 12) shows the distribution of the collected apps into the available categories in 2015 and 2019. To answer RQ1, RfQ2, RQ3 and RQ4, we consider only apps from 2019. Instead, to answer RQ5, we use apps from both years 2015 and 2019.

### 5.4. Analysis procedure

We launched `ATADetector` on the subjects apps in a High-Performance Computing (HPC) cluster available in Fondazione Bruno Kessler (FBK). In this way, we could run several threads in parallel to complete the analysis faster. On average, `ATADetector` analyzes one app per minute. The overall analysis took around two weeks.

`ATADetector` produces two reports for each analyzed app, i.e. a long and a short version of the results of the analysis. The longest one is more detailed and it contains the protection atoms described in Section 3 along with the number of times `ATADetector` detected each protection atom in the app. The shorter one is more general and it reports, for each fingerprint, whether `ATADetector` identified the related protection in the app. In Fig. 5 (page 13), we present an example of a short report. The short report lists the protections and states whether `ATADetecor` identified the protection, indicated with the number 1, or not, indicated with the number 0. For each protection, the short report specifies whether it is at `Java` or at `Native` level. For the former, the short report furtherly specifies whether `ATADetector` identified the protection in third-party libraries (`Java_1`) or in the app developers' code (`Java_2`).

We present the analysis results as barplots, commenting on the trends that are evident in the graphs. Moreover, when comparing trends for protections RQ3 (developers' code Vs libraries code),



(a) Apps collected per Category in 2015

(b) Apps collected per Category in 2019

**Fig. 4.** Apps Collected per Category in 2015 (4a) and 2019 (4b).

```
{
    'SignatureChecking_Java'                          : 0,
    'SignatureChecking_Java_1'                        : 1,
    'SignatureChecking_NATIVE'                        : 0,
    'CodeIntegrityChecking_Java'                      : 0,
    'InstallerVerification_Java'                      : 0,
    'InstallerVerification_Java_1'                    : 1
    'SafetyNetAttestation_Java'                       : 0,
    'EmulatorDetection_Java'                          : 0,
    'EmulatorDetection_Java_1'                        : 1,
    'EmulatorDetection_NATIVE'                        : 0,
    'DynamicAnalysisFrameworkDetection_Java'          : 0,
    'DynamicAnalysisFrameworkDetection_NATIVE'        : 0,
    'DebuggerDetection_Java'                          : 0,
    'DebuggerDetection_Java_1'                        : 1,
    'DebuggerDetection_NATIVE'                        : 0,
    'DebuggableStatusDetection_Java'                  : 0,
    'DebuggableStatusDetection_NATIVE'                : 0,
    'AlteringDebuggerMemoryStructure_NATIVE'          : 0,
}
```

**Fig. 5.** Short Report Produced by `ATADetector` for *com.cashback.card*.

RQ4 (`Java` Vs `Native`) and RQ5 (2015 Vs 2019), we need to assess whether any observed difference is statistically significant and not due to random variation. To analyze whether this difference is significant, we use the Fisher's exact test [21], more accurate than the $\chi^2$ test, which is another possible alternative to test the presence of differences in categorical data. In this statistical test, we consider a 95% significance level, i.e. we accept a 5% probability of committing a Type I error.

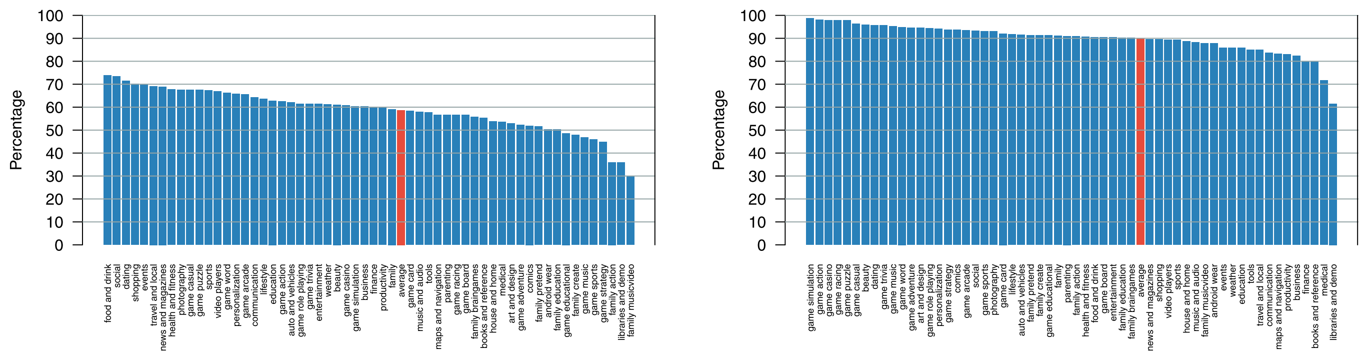
## 5.5. RQ1 - Adoption of AD and AT protections

Since the categories do not contain the same number of apps, we measure the relative percentage and not the absolute number. Fig. 6 (page 13) shows the results of this aggregation.

On average (red columns), 90% of top category Android apps implement AT protections and 58.69% of apps implementing AD protections. The most protected categories are Games, Dating and Social while the less protected are related to Family and Libraries. Surprisingly, also the Medical category is among the ones less protected. We can infer that developers are more inclined to deploy AT rather than AD protections and that the vast majority of apps is equipped with AD and AT protections.

Furthermore, we examine how many times `ATADetector` identified the protections singularly. For each short report related to an app, we count the detected protections summing these occurrences in Fig. 7 (page 14)

The most deployed protection is *Signature Checking* with 88.80% Android apps implementing it. Then, there are *Installer Verification* (74.42%) and *Emulator Detection* (49.83%) protections. The least deployed protections are *Debuggagle Status Detection* (2.02%), *Code Integrity Checking* (1.00%) and *Altering Debugger Memory Structure*, never detected in the analyzed apps. The last is a particularly complicated protection to be implemented at `Native` level. Therefore,



(a) Percentage of Apps Implementing at Least One AD Protection

(b) Percentage of Apps Implementing at Least One AT Protection

**Fig. 6.** Percentage of apps implementing at least one AD (6a) and AT (6b) protection.

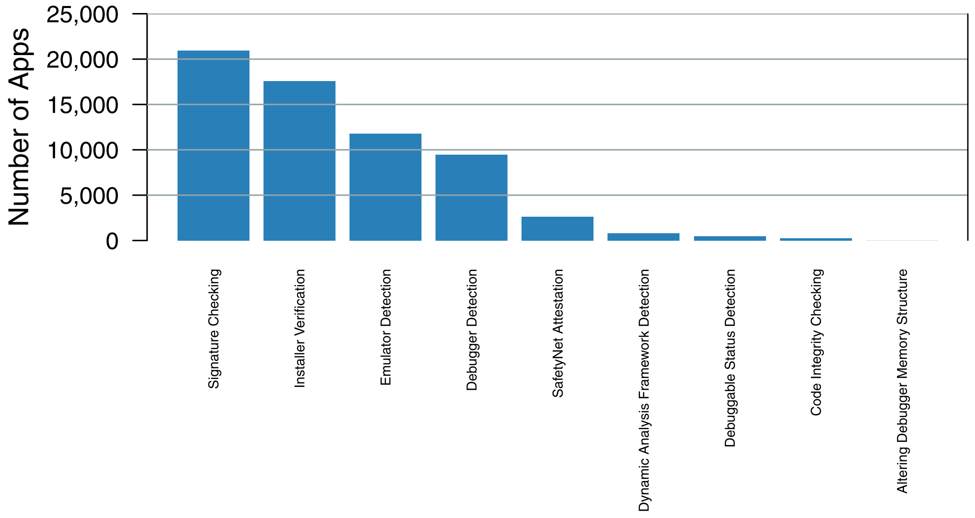


**Fig. 7.** Number of apps adopting the related protections.

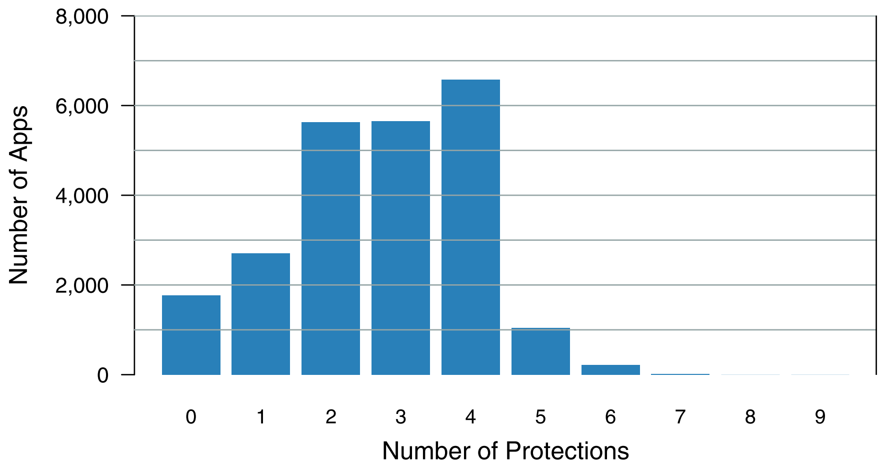


**Fig. 8.** Apps divided per number of protections implemented.

we can suppose that (i) few developers deployed it and (ii) they took care of hiding it (e.g., through Obfuscation).

### 5.6. RQ2 - Integration of multiple protections

We now examine the overall number of protections an app implements. Note that we do not differentiate by category or type of protection (i.e. AD or AT). Therefore, we count the detected protections reported in the short versions of the reports and sum them in Fig. 8 (page 15).

There are 1,769 apps out of 23,610 apps (7.49%) that implement no protections, while the vast majority usually implements two (5,630 or 23.84%), three (5,653 or 23.94%) or four protections (6,575 or 27.84%). Apps implement three protections on average. From the statistics, we can infer that developers are likely to deploy more than one protection.

We also analyzed how each protection integrates with others. For each pair of protections, regardless of the scope and level, we counted the occurrences `ATADetector` detected it. Table 6 (page 16) summarizes the results of this analysis. Each cell contains the number of times `ATADetector` identified the two protections together.

The most popular pair is *Signature Checking* and *Installer Verification* protections with 17,329 apps implementing both of them. The second is *Signature Checking* with *Emulator Detection* protections with 11,203. Indeed, these three protections are also the ones most employed by developers. In general, Table 6 (page 16) accurately reflects the statistics presented in Fig. 7 (page 14).

### 5.7. RQ3 - Protections in developers' code and third-party libraries

The results that we presented so far suggests that AD and AT protections are quite popular among Android apps, given that most of the apps deploy at least one protection. Their developers employ both AD and AT protections and even more protections at the same time. However, we want to investigate whether the protections come from the developers of the apps or derive from third-party libraries. We collect the names of the packages of many of the most used third-party libraries. Moreover, we search online for similar libraries and collect their package names too. In total, we collect 83 library packages. The complete list is reported in Appendix D.

We empirically observed that apps are likely to retain the names of the `Java` packages even though the apps are obfuscated. Wang Yan et al. [22] found that ProGuard[11] is the most widely used tool to obfuscate Android apps, while Wermke

[11] https://www.guardsquare.com/en/products/proguard.

**Table 6**
Count of how many times two protections are deployed together.

| | Code Integrity Checking | Installer Verification | SafetyNet Attestation | Emulator Detection | Dynamic Analysis Framework Detection | Debugger Detection | Debuggable Status Detection | Altering Debugger Memory Structure |
|---|---|---|---|---|---|---|---|---|
| Signature Checking | 153 | 17,329 | 2,628 | 11,203 | 796 | 8,979 | 474 | 0 |
| Code Integrity Checking | | 131 | 7 | 102 | 6 | 78 | 1 | 0 |
| Installer Verification | | | 2,413 | 10,698 | 759 | 8,026 | 451 | 0 |
| SafetyNet Attestation | | | | 1,065 | 99 | 597 | 94 | 0 |
| Emulator Detection | | | | | 679 | 7,159 | 464 | 0 |
| Dynamic Analysis Framework Detection | | | | | | 524 | 199 | 0 |
| Debugger Detection | | | | | | | 229 | 0 |
| Debuggable Status Detection | | | | | | | | 0 |

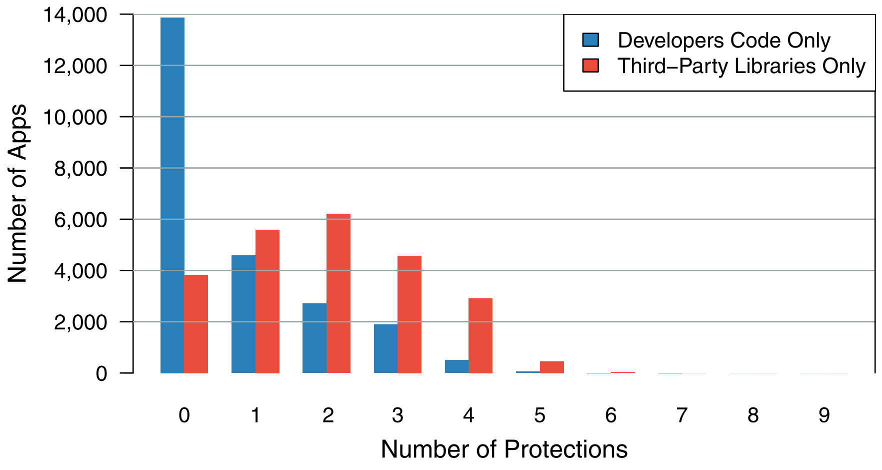


**Fig. 9.** Apps Divided per Number of Protections Implemented in the Developers Code and Third-Party Libraries Code.

et al. [23] found that the vast majority of Android app developers fails to correctly configure ProGuard. Since developers have to configure ProGuard to obfuscate third-party libraries explicitly, we can suppose that these are the reasons why we observed many not-obfuscated `Java` package names. Being so, we can distinguish between third-party libraries and developers' code in the app. Consequently, we can understand where `ATADetector` identified the protections. Fig. 9 (page 17) reports the results of this analysis.

Only 28% (17,979 over 63,858 identified protections) of the protections come from the developers, while the remaining 72% (45,879 over 63,858 identified protections) derive from third-party libraries. Unexpectedly, we notice that most of the detected protections derive from third-party libraries.

Fig. 9 (page 17) shows how many protections are implemented on each app directly in the developers' code (blue bars) or in the libraries code (red bars). Most of the Android apps (13,867 over 23,610) contain no protection in developers' code. Among the remaining, 4,588 apps contain just one protection and 2,705 apps contain two protections in the developers' code. The trend seems different for libraries code. In fact, most the apps (i.e. 6210 apps) contain two protections in the libraries code, while 5,589 and 4,577 apps contain, respectively, 1 and 3 protections in the libraries code. Only 3,830 apps contain no protection in the libraries code.

To assess if the difference in the observed trends is statistically significant, we use the Fisher's exact test, and the resulting p-value is $< 0.001$. Considering that the p-value is below 5%, we can conclude that the difference in the observed distribution of protections in developers code and library code is statistically significant and not just due to random errors.

We also investigate which kind of protections third-party libraries implement and report the results in Fig. 10 (page 18). By comparing these results with the results of RQ1 reported in Fig. 7 (page 14), we can speculate that there is no substantial difference between the protections chosen by app developers and libraries developers. Indeed, the most deployed protection is still *Signature Checking* with 65.45% of Android apps including third-party libraries that implement it, followed again by *Installer Verification* (51.29%). The only difference is that third-party libraries developers prefer to implement *Debugger Detection* (32.45%) rather than *Emulator Detection* (31.67%) protections.

Concerning protections derived from third-party libraries, the vast majority of apps (19,780 over 23,610 apps) employs libraries with at least one protection. Unfortunately, these protections do not cover the logic of the app but only the functioning of the library itself. Therefore, their effectiveness reduces to that scope only.

### 5.8. RQ4 - Protections deployed at `Java` and `Native` level

Another important aspect is the ratio between protections implemented at `Java` and `Native` levels. We examine it by considering the number of protections identified at these two different levels. Fig. 11 (page 19) reports the results of this comparison.

Considering the protections implemented at `Java` level (blue bars), we see many apps with 2, 3 and 4 protections, while very few apps have no `Java` protection. Conversely, if we consider protections implemented at `Native` level (red bars), we see that the majority of the apps have no `Native` protection. Only a few apps have 1 or more protections at the native level. According to the result of the Fisher's exact test, this difference in the trends of `Java` and `Native` protections is statistically significant (p-value $< 0.05$).

We observe that 99% of the identified protections are implemented in `Java`. Only 2.2% (521 over 23,610 apps) of top-category Android apps implement `Native` protections, and in general no more than one. We can infer that there are more protections

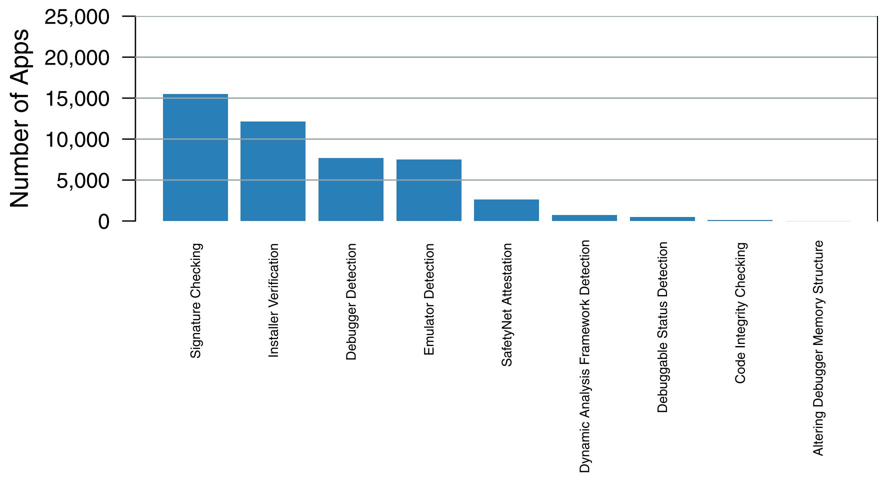


**Fig. 10.** Number of apps containing the related protections in third-party libraries only.

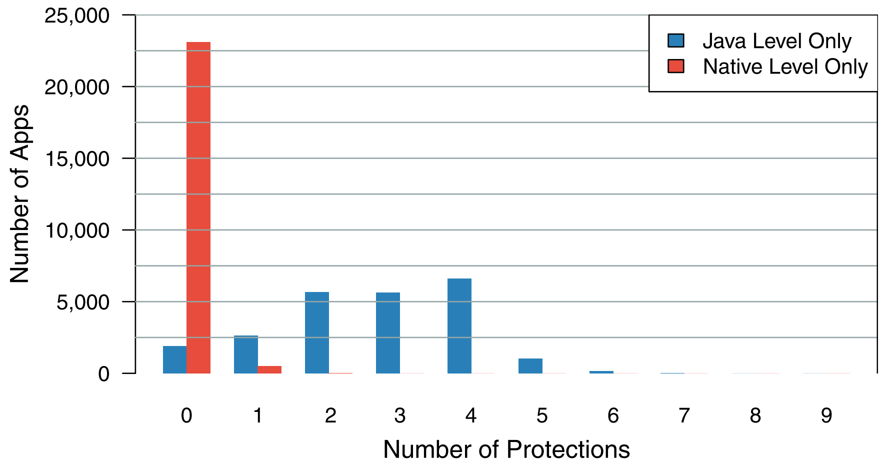


**Fig. 11.** Apps divided per number of protections implemented at `java and native` level.

deployed at the Java level rather than at the Native one, implying that developers quite never consider implementing protections in C++. Then, the trend of Java protections is practically equal to the one presented in Fig. 8 (page 15). There can be several explanations for this lack of Native protections:

- It is more difficult to implement protections at the Native level than at the Java one [9].
- While every app contains Java code, not all apps include Native code.
- Empirically, we noticed that it is easier to find on the internet snippets of code for Java protections rather than Native ones.

### 5.9. RQ5 - Evolution in adoption of AD and AT protections

This last research question compares statistics about identified protections between the datasets of apps from 2015 and 2019.

The results are reported in Fig. 12 (page 20). Results are in percentage, because of the different number of apps in the two datasets of this analysis. According to Fig. 12 (page 20), apps in 2019 seem to deploy more protections than apps from 2015. In fact, the percentage of adoption increases from 80.50% to 88.80% for the *Signature Checking* protection, from 71.46% to 74.42% for the *Installer Verification* protection and from 41.21% to 49.83% for the *Emulator Detection* protection. The other protections follow a similar pattern, with the only exception of *SafetyNet Attestation*, whose adoption rate decreases from 12.41% to 11.13% and the *Code Integrity Checking*, that decreases from 0.90% to 0.69%.

We applied the Fisher's exact test on the data in Fig. 12. The test result confirms that the different trends between 2015 and 2019 are statistically significant.

## 6. Discussion

In this section, we discuss technical limitations and the threats to validity.

### 6.1. Technical limitations

ATADetector is a static analysis tool and Java Reflection, even though mitigated by detecting FQN, will always be a limitation. Through Reflection, developers can screen API invocations in the code of their apps. Besides, there are methods through which developers can furtherly hinder the analysis of their apps. For instance, String Encryption consists in encrypting constant strings to make them unreadable by static analysis tools. Then, a routine decrypts the strings at runtime when needed. ATADetector relies on strings for both the detection of FQN to mitigate Reflection and as protection atoms themselves. Therefore, String Encryption threatens the effectiveness of the detection of our tool as well since it makes statically reading the value of these strings nearly impossible.

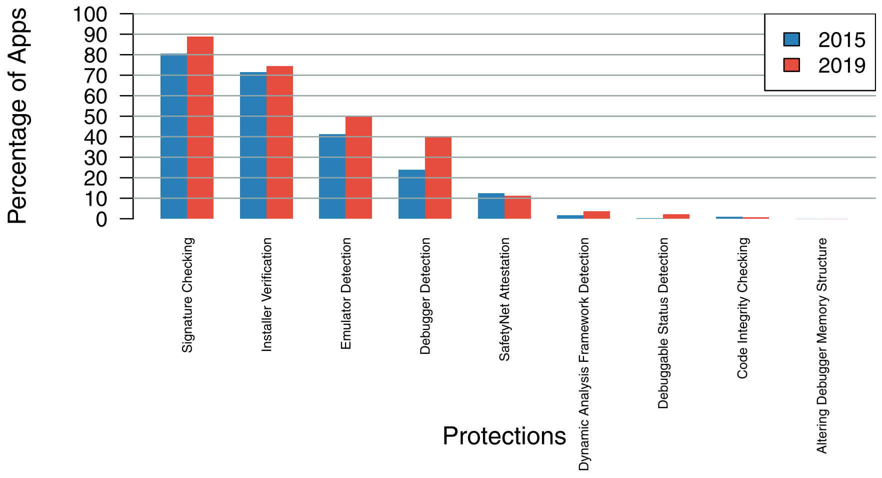


**Fig. 12.** Percentage of All Apps in 2015 and 2019 Adopting the Related Protection.

In our large-scale study, we considered apps written in `Java` only. Still, `Java` is not the only option available to Android developers. For instance, *Cordova*[12] is an open-source framework for apps development by Apache. It exploits standard web technologies like *HTML5, CSS3* and *JavaScript* for cross-platform deployment. Therefore, developers can publish *Cordova* apps both on iOS and Android. Even though our approach may be valid also for apps written with web technologies, our fingerprints are not. However, we can argue that the vast majority of Android developers implement their apps with `Java`.

### 6.2. Threats to validity

*Construct validity:* There are three reasons for which the statistics we produced may not be accurate:

- Our fingerprints may not cover all possible ways in which developers can implement AT or AD protections. Despite we adopted an incremental refinement and validation of our fingerprints to limit this threat, there may be other programming elements we did not consider that developers can use to implement their protection. Therefore, we could still have missed some protections.
- Third-party libraries detection is based on a list that may not contain the package names of all third-party libraries available to Android developers. Therefore, this measurement could not be extremely accurate. Also, if the developers obfuscated the package names of the libraries, we would misclassify some protections. However, note that solving these issues would only lower even more the percentage of protections found in the apps developers' code. Detecting third-party libraries with more precision would categorize more protections as belonging to the libraries themselves. Therefore, the percentage of protections coming from the developers is anyway less than 28%.
- In general, it is known that code hardening (e.g., reflection, string encryption, obfuscation) causes problems to static analysis. Therefore, `ATADetector` may have missed some protections because concealed by the developers. However, `ATADetector` found more protection in 2019 than in 2015. This suggests that, even though apps from 2019 may have been hardened, our approach works well on modern and obfuscated apps.

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

*External validity:* Our analysis considers apps from one single app store only, i.e. the Google Play Store. Therefore, our analysis could be biased and our results might not extend in general to apps coming from other stores. To limit this threat, we considered apps from all the categories to have a wide variety of cases, and from different years.

## 7. Related work

In the literature, numerous researchers focused on the security analysis of Android apps. Concerning protections against malicious reverse engineering, some works presented novel approaches and schemes. Others described large-scale studies to assess several properties of Android apps related to security and reverse engineering.

### 7.1. New protections for android apps

Piao et al. [5] proposed a server-based approach to provide both encryption and AT protection. A server stores the main functionalities of the app encrypted together with a tamper detection protection. When needed, the code is downloaded and decrypted with a one time secret key. Similarly, Viticchie et al. [24] automatized the deployment of AT protections using a *Reactive Remote Attestation* technique. This technique consists in splitting the code of the app and moving the core routines server-side. Before accessing these routines, the app has to prove its integrity to the server. In a successful scenario, the server executes the core routines and returns the results to the app. Otherwise, the server does not run the code to prevent the tampered app from continuing its execution. Divilar is a tool developed by Zhou et al. [6] for re-encoding an Android app with a random instruction set over `dex` bytecode as an AT protection. The app executes with a specialized virtual instruction interpreter, designed to be integrated with the Dalvik virtual machine to reduce the performance overhead. Wan et al. [25] developed an AD protection by building check points for integrity verification. They analyzed open-source tools for hookings methods and APIs to identify such check points. If one of these tools hooks a method to debug an app, it will alter the value of the check points related to the method. Their approach can detect this modification and then raise a warning. Abrath et al. [26] investigated the weaknesses in AD protection through self-debugging. They argue that attackers can easily bypass this approach with little effort. Therefore, they propose a new technique in which portions of the code of the app are moved in the debugger itself, hindering the attackers in the reverse engineering process. They also provided an implementation with a prototype and validated their technique with penetration testers. Jing et al. [27] proposed Morpheus, a framework for Android emulator detection. Their approach consists of analyzing Android system artefacts observable by Android apps, and then generate heuristics to detect Android emulators automatically. Such heuristics were tested against both Android emulators and real devices, obtaining high accuracy. Other works propose synergy between protections. Since hooking APIs is an efficient way to bypass AD protections, Kyeonghwan et al. [28] combined a simple AD protection (i.e. checking the value of the *debuggable* flag in the Android manifest) with the detection of API method hooking attacks. Vasileiadis [29] collected both AD and AT protections in a comprehensive approach into evaluating the state of an Android app and its running environment. The collection includes, inter alia, signature verification, debugger and emulator detection and remote attestation.

### 7.2. Large-scale studies on android apps

Ghafari et al. [30] analyzed thousands of app from the Google Play store to detect pre-defined patterns of coding errors that lead to security vulnerabilities. First, they defined a list of bad programming habits, the resulting vulnerability and a possible solution or mitigation. Then, they implemented the detection of such errors in a static analysis tool. They asserted that more than 90% of the examined Android apps contain at least one potential vulnerability. Shan et al. [31] focused on the categorization of what they defined as “self-hiding behaviours” in Android apps. These techniques allow apps to conceal their activities from end-users. They provided a list of such behaviours along with a description and an example implementation. From this code, they extracted unique patterns and designed a static analysis algorithm able to detect them. One of their findings is that legitimate apps employ these behaviours as much as malicious ones. Gao et al. [32] analyzed trends in misuse of Android’s cryptographic-related APIs. A misuse is defined as a wrong or insecure configuration of such APIs, like the use of outdated algorithms (e.g., MD5), the hardcoding of salt values and the storing of sensitive data as (immutable) Java strings. The initial assumption is that app updates across an app lineage are likely to fix these misuses. The authors employed an already existing static analysis tool in a large-scale study on 40 thousands of apps lineages. Counterintuitively, the finding is that misuses of crypto-APIs are not likely to be fixed by app developers. Habchi et al. [33] performed a large-scale study to analyze bad programming practices, which they call “code smells”, on Android apps. Their goal is to understand whether these smells come from inexperienced developers only. The authors defined and described 8 bad programming practices and build on top of them a static analysis tool. Their finding is that smells are not the responsibility of an isolated group of developers, and there are no distinct groups of code smell introducers and removers. Developers who introduce and remove code smells are mostly the same.

Even though with a different purpose, these works proposed an approach similar to ours. They identified specific patterns and exploited static analysis to detect them in Android apps. However, other works specifically targeted the adoption of protections against malicious reverse engineering in Android apps, either to assess their implementation rate or to conduct derivative studies.

### 7.3. Large-scale study on the adoption of protections in android apps

Wermke et al. [23] investigated the extent to which Obfuscation is used in Android apps. They exploited static analysis considering identifiers like package names, classes, methods and fields. They aimed at detecting Obfuscation by Proguard.[13] They tested their algorithm on manually protected open source apps from F-Droid.[14] Finally, they launched a large-scale analysis on more than a million Android apps from the Google Play store, finding that only 24.92% of apps are obfuscated bu the developers. Kaur et al. [34] tackled the task of Obfuscation identification from a different and novel approach, exploiting spatial analysis. This technique investigates patterns present in images calculated directly from binary files. The authors created grey-scale images from Android APKs and then calculated first- and second-order statistics like the Shannon Entropy and Chi-Square. They were able to achieve a significant accuracy (nearly 90%) in fingerprinting Obfuscation tools together with their configuration. Wang Yan et al. [22] exploited Machine Learning techniques to study and classify Obfuscation in Android apps. Their purpose was to distinguish whether an app is obfuscated or not and what tool the developers employed. They employed several tools to create different obfuscated versions of open-source apps downloaded from the F-Droid repository. After defining and tuning their classifiers, they performed a Large-Scale analysis of Google Play apps to study the percentage of obfuscated apps and the most frequent tools. They managed to identify the configuration of the tools with more than 90% accuracy. Wang Pei et al. [35] studied the deployment of Obfuscation techniques on the Apple Store apps. Their purpose was to discover to what extent iOS developers employ this protection. For each app, they assessed the amount of protected code discerning third-party libraries. Eventually, they tested the resilience of the Obfuscation techniques on a set of apps. Despite an increasing trend of the usage of such protection, they found that many apps are still vulnerable to low-effort reverse engineering.

The literature presents several studies on protections against malicious reverse engineering and large-scale studies on Android apps. As we discussed, many researchers proposed an approach similar to ours. First, they identified peculiar patterns, analogue to our protection atoms. Then, they tuned the patterns on toy apps. Once automatized the process, they started a large-scale study on apps. However, regarding protections against attackers, all of these studies focused on Obfuscation identification only. They did not consider other kinds of protections against malicious reverse engineering. To the best of our knowledge, we are the first to assess the adoption rate of AD and AT protections in Android apps.

## 8. Future work

Several interesting areas can be investigated to enhance the large-scale analysis we presented:

- `ATADetector` does not consider the context in the detection of the protections. It identifies each protection atom separately and then it consults the fingerprint. Instead, it would be interesting to introduce a context in the extraction of protection atoms. We could track a particular protection atom to see when and how the developers used it. For instance, we could check whether the package name of an app store and the value returned by the `getinstallerpackagename` API are the parameters of a `.equals` method. In this way, we would obtain very accurate detections by removing many false positives
- Since not strictly related to AD or AT, we excluded some protections from our analysis, like the *Root Detection* protection. Indeed, this protection focuses on the status of the smartphone rather than on eventual tampering on the app. However, it would be interesting to investigate the adoption of

[13] https://www.guardsquare.com/en/products/proguard.
[14] https://f-droid.org/.

this protection also. Similarly, there may be other protections worth considering.

- ATADetector detects the protections by identifying the protection atoms through static analysis. We chose to exploit static analysis since it was the natural automation process for our extraction of protection atoms. However, we can automatize the detection with other techniques and check whether they perform better or not. Indeed, there are different approaches for the actual implementation:
  - *Machine learning*: The protection atoms we defined can work as features for training the model. The challenge is to produce a training set large enough to train the model.
  - *Dynamic analysis*: This approach would overcome Reflection and String Encryption. However, an app could activate certain protections under certain particular conditions only. For instance, it could run AD protections after the login or AT protections when a free trial of eventual premium features expires. Therefore, it would be difficult to tell whether there are no protections or the analysis was not thorough enough.
  - *Spatial analysis*: Kaur et al. [34] employed this interesting kind of analysis for Obfuscation detection in Android apps. However, we have to understand whether the protection atoms we are interested in are too small to be accurately detected in the generated images or not.

## 9. Conclusion

In this paper, we described the first large-scale study about the detection of AD and AT protections in Android apps. Our purpose is to understand the extent to which Android app developers employ these protections. We identified and described nine different protections against malicious reverse engineering. We collected example implementations and extracted peculiar protection atoms, both at Java and Native levels, producing and refining the fingerprints. We developed a tool, ATADetector, to automatize the detection task. Before launching the large-scale analysis, we tuned the fingerprints with three incremental validation steps to achieve more accurate detection rates. Finally, we analyzed 37,783 Android apps.

We defined five research questions and four metrics. We analyzed the percentage of protected apps by category and how frequents protections integrate each other. Then, we investigated whether the detected protections came from third-party libraries or not. We compared the ratio of protections implemented at Java and Native levels and then assessed the evolution of the adoption of the protections between 2015 and 2019.

At first, the results seemed to indicate that a high percentage of apps deploy AD and AT protections. Almost all apps implement AT protections and around two out of three implementing AD protections. Furthermore, an app contains 3 protections on average. However, we discovered that only 28% of all protections come from apps developers, while the remaining derive from third-party libraries. Therefore, the vast majority of protections do not provide any defence against attacks to the logic of the app. We also found that the ratio between Java and Native level protections is of 99 to 1. This implies that developers implement almost all protections in Java that attackers can more easily reverse and bypass respect to Native protections. Furthermore, we observed that apps from 2019 generally employ more protections than apps from 2015.

Attackers analyze and tamper Android apps to unlock premium features, insert malware and redirect ads revenue. Even though it is not possible to definitively block malicious reverse engineering, app developers can hinder the process by securing the code through the use of AD and AT protections. Our findings show that Android apps are not as protected as they could be. This result is even more serious since we considered top-category apps.

The final reports and aggregated results can be found, together with ATADetector and other material, in our GitHub repository [36].

## Declaration of Competing Interests

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Appendix A. Protections implementation collection

This appendix contains the example implementations for the protections we identified. First, we divide between AD and AT protections. Then, for each protection, we describe the implementation and report the Java and the Native code, when present. Additional information can be found in the related references.

### A1. AD protections

- *Emulator detection* (Java implementation in Fig. A1, page 25): an app can obtain the properties of the smartphone in several ways. it can access the android.os.Properties class through Reflection [11] (lines 22-37 Java), query the Build class (lines 2-19 Java) [9] or execute the getprop command (lines 40-44 Java) [37], both at Java and at Native level. It can also check the presence of emulator related files [13] (lines 47-51 Java).
- *Dynamic analysis framework detection* (Java implementation in Fig. A2, page 26, Native implementation in Fig. A3, page 27): The simplest way to detect a dynamic analysis framework is to scan package names, files or binaries to look for resources known to be components of these frameworks. An app can throw an exception and check whether Xposed is present in the stack trace [9,11] (lines 2-27 Java). It can also iterate through the list of running processes to check whether the Frida server is running [38] (lines 30-45 Java). At Native level, an app can ping the TCP port 27047, used by the Frida server as default, to see whether it is open [38] (lines 1-11 Native). Also, it can also check if Frida-related libraries are mapped into memory [38] (lines 13-28 Native).
- *Debugger detection* (Java implementation in Fig. A4, page 27, Native implementation in Fig. A5, page 28): an app can discover the presence of a JDWP debugger through the Debug.isDebuggerConnected API (lines 1-3 Java). The app can invoke the same API through the gDvm structure at Native level (lines 1-6 Native) [9,11]. An app can detect the GDB debugger by checking if there are processes attached to the process of the app by reading the *TracerPid* value in the */proc/self/status* file (lines 5-22 Java) [37]. Beside reactive protections, there are also preventive ones. For instance, an app can attach a mock debugger process to itself so to prevent a real debugger process from functioning properly (lines 8-30 Native) [39]. For what concerns a GDB debugger, remember that the best protection is to prevent it from attaching to the process of the app.
- *Debuggable status detection* (Java implementation in Fig. A6, page 28): The attackers, to allow JDWP debugging, have to alter the value of the debuggable flag in the manifest of the app. In this way, the Android operating system starts an extra thread for handling the JDWP protocol. An app can access and check the value of this flag either through the ApplicationInfo.FLAG_DEBUGGABLE

```java
// check properties from Build class
private boolean hasEmulatorBuildProp() {
      return Build.FINGERPRINT.startsWith("generic")
              || Build.FINGERPRINT.startsWith("unknown")
              || Build.MODEL.contains("google_sdk")
              || Build.MODEL.contains("Emulator")
              || Build.MODEL.contains("Android SDK built for x86")
              || Build.MANUFACTURER.contains("Genymotion")
              || (Build.BRAND.startsWith("generic") && Build.DEVICE.startsWith("generic"))
              || Build.PRODUCT.contains("google_sdk")
              || Build.HARDWARE.contains("goldfish")
              || Build.HARDWARE.contains("ranchu")
              || Build.BOARD.contains("unknown")
              || Build.ID.contains("FRF91")
              || Build.MANUFACTURER.contains("unknown")
              || Build.SERIAL == null
              || Build.TAGS.contains("test-keys")
              || Build.USER.contains("android-build");
}

// method to get properties of Properties class through Reflection
private static String getProp(Context ctx, String propName) throws Exception {
  ClassLoader cl = ctx.getClassLoader();
  Class<?> class = cl.loadClass("android.os.properties");
  Method getProp = class.getMethod("get", String.class);
  Object[] params = {propName};
  return (String) getProp.invoke(class, params);
}

// check properties from the Properties class
private boolean hasQemuBuildProps() {
  return    "goldfish".equals(getProp(context, "ro.hardware"))
        ||  "ranchu".equals(getProp(context, "ro.hardware"))
        ||  "generic".equals(getProp(context, "ro.product.device"))
        ||  "1".equals(getProp(context, "ro.kernel.qemu"))
        ||  "0".equals(getProp(context, "ro.secure"));
}

// check properties from the getProp command
private String getSystemProperty(String propertyName) throws Exception {
  Process getPropProcess = Runtime.getRuntime().exec("getprop " + propertyName);
  BufferedReader osRes = new BufferedReader(new InputStreamReader(getPropProcess.getInputStream()));
  return osRes.readLine();
}

// check the presence of pipes related to emulators
private static boolean hasQemuFile() {
        return new File("/init.goldfish.rc").exists()
                || new File("/sys/qemu_trace").exists()
                || new File("/system/bin/qemud").exists();
}
```

**Fig. A1.** Java example implementation of *Emulator Detection* protection.

(lines 2-4 Java) or the BuildConfig.DEBUG (lines 5-7 Java) attribute [11].

- *Altering debugging memory structure (*Native*implementation in Fig. A7, page 29)*: An app can tamper with the variables related to the JDWP debugger to hinder its correct functioning. In *Dalvik*, the app can modify the pointers of the DvmGlobals structure thorugh the global variable gDvm (lines 1-3 Native) [9]. In *ART*, the app can do the same by e by overwriting JDWP method pointers (lines 5-40 Native) [9]. For instance, an app can overwrite the address of the function jdwpAdbState::ProcessIncoming with the address of JdwpAdbState::Shutdown. In this way, the debugger will disconnect immediately when a new process is coming.

## A2. AT protections

- *Signature checking* (Java implementation in Fig. A8, page 29, Native implementation in Fig. A9, page 30): A tampered app does not have the same digital signature anymore. Therefore, an app can compare the current signature of the APK file with the original one. The app can implement this protection both at Java and Native level. In the former case, the app can obtain the current signature through dedicated APIs using the PackageManager.GET_SIGNATURES and the PackageInfo.signatures (API $<$ 28) or the PackageManager.GET_SIGNING_CERTIFICATES and the PackageInfo.signingInfo (API $\geq$ 28) APIs (lines 1-16 Java) Alexander-Bown. In the latter case, the app can extract and parse the *CERT.RSA* file [12].
- *Code integrity checking* (Java implementation in Fig. A10, page 30): Similarly to the *Signature Checking* protection, an app can compute a digest value on a resource or file and then compare it with the expected one. Therefore, an app could access and hash the file containing the Java code (i.e. the *.dex* file) and check whether this value is the original one or not. App developers can use standard libraries like "Zipentry"[15] to automatically obtain useful values like the CRC code (lines 1-7 Java) [9].

[15] https://developer.android.com/reference/java/util/zip/ZipEntry.

```java
// Xposed detection through exception stack strace
try {
  throw new Exception();
}
catch(Exception e) {
  int zygoteInitCallCount = 0;
  for(StackTraceElement stackTraceElement : e.getStackTrace()) {
    if(stackTraceElement.getClassName().equals("com.android.internal.os.ZygoteInit")) {
      zygoteInitCallCount++;
      if(zygoteInitCallCount == 2) {
      Log.wtf("HookDetection", "Substrate is active on the device.");
      }
    }
    if(stackTraceElement.getClassName().equals("com.saurik.substrate.MS$2") &&
      stackTraceElement.getMethodName().equals("invoked")) {
      Log.wtf("HookDetection", "A method on the stack trace has been hooked using Substrate.");
    }
    if(stackTraceElement.getClassName().equals("de.robv.android.xposed.XposedBridge") &&
      stackTraceElement.getMethodName().equals("main")) {
      Log.wtf("HookDetection", "Xposed is active on the device.");
    }
    if(stackTraceElement.getClassName().equals("de.robv.android.xposed.XposedBridge") &&
      stackTraceElement.getMethodName().equals("handleHookedMethod")) {
      Log.wtf("HookDetection", "A method on the stack trace has been hooked using Xposed.");
    }
  }
}

// check if Frida server is running
public boolean checkRunningProcesses() {
  boolean returnValue = false;
  List<RunningServiceInfo> list = manager.getRunningServices(300);
  if(list != null){
    for(int i=0;i<list.size();++i){
      if(list.get(i).process.contains("fridaserver")) {
        returnValue = true;
      }
    }
  }
  return returnValue;
}
```

**Fig. A2.** `Java` implementation of *Dynamic Analysis Framework Detection* protection.

```c
boolean is_frida_server_listening() {
  struct sockaddr_in sa;
  memset(&sa, 0, sizeof(sa));
  sa.sin_family = AF_INET;
  sa.sin_port = htons(27047);
  inet_aton("127.0.0.1", &(sa.sin_addr));
  int sock = socket(AF_INET , SOCK_STREAM , 0);
  if (connect(sock , (struct sockaddr*)&sa , sizeof sa) != -1) {
    /* Frida server detected */
  }
}

boolean is_frida_library_loaded() {
  char line[512];
  FILE* fp;
  fp = fopen("/proc/self/maps", "r");
  if (fp) {
    while (fgets(line, 512, fp)) {
      if (strstr(line, "frida")) {
        /* Evil library is loaded */
      }
    }
    fclose(fp);
  }
  else {
    /* Error opening /proc/self/maps. If this happens, something is off. */
  }
}
```

**Fig. A3.** `Native` implementation of *Dynamic Analysis Framework Detection* protection.

- *Installer verification* (`Java` implementation in Fig. A11, page 31): Usually, attackers publish tampered and repackaged apps in third-party app stores [15,40]. When installing an app, the Android operating system keeps track of the source of the APK file. This value is available through the `PackageManager` method `getInstallerPackageName`. In particular, this returns the package name of the app through which the end-user installed the current app. An app can obtain this value and check whether it is consistent with the app

```
public boolean isJDWPDebuggerConnected() {
    return Debug.isDebuggerConnected();
}

public static boolean isGDBDebuggerConnected() throws Exception {
  BufferedReader reader = null;
  reader = new BufferedReader(new InputStreamReader(new FileInputStream("/proc/self/status")), 1000);
  String line;
  while ((line = reader.readLine()) != null) {
    if (line.length() > tracerpid.length()) {
      if (line.substring(0, tracerpid.length()).equalsIgnoreCase(tracerpid)) {
        if (Integer.decode(line.substring(tracerpid.length() + 1).trim()) > 0) {
          return true;
        }
      break;
      }
    }
  }
  reader.close();
  return false;
  }
}
```

**Fig. A4.** `Java` implementation of *Debugger Detection* protection.

```
boolean is_debugger_connected {
  if (gDvm.debuggerConnected || gDvm.debuggerActive) {
    return JNI_TRUE;
  }
  return JNI_FALSE;
}

child_pid = fork();
if (child_pid == 0) {
  int ppid = getppid();
  int status;
  if (ptrace(PTRACE_ATTACH, ppid, NULL, NULL) == 0) {
  waitpid(ppid, &status, 0);
  ptrace(PTRACE_CONT, ppid, NULL, NULL);
    while (waitpid(ppid, &status, 0)) {
      if (WIFSTOPPED(status)) {
        ptrace(PTRACE_CONT, ppid, NULL, NULL);
      }
      else {
      // Process has exited
      _exit(0);
      }
    }
  }
}
else {
  pthread_t t;
  /* Start the monitoring thread */
  pthread_create(&t, NULL, monitor_pid, (void *)NULL);
}
```

**Fig. A5.** `Native` implementation of *Debugger Detection* protection.

```
public boolean isDebuggable() {
  if ((context.getApplicationInfo().flags & ApplicationInfo.FLAG_DEBUGGABLE) != 0) {
    return true;
  }
  else if (BuildConfig.DEBUG) {
    return true;
  }
  else {
    return false;
  }
}
```

**Fig. A6.** `Java` implementation of *Debuggable Status Detection* protection.

stores where the developers published the app (lines 1-6 `Java`) [11]. Suppose the developers published their app only in the Google Play Store. Therefore, end-users should have installed the app through the Play Store app that has "com.android.vending" as the package name. If the value returned by the `getInstallerPackageName` API is "cm.aptoide.pt", the app was installed from Aptoide,[16] an independent Android app store. Therefore, some attackers likely tampered the app.

[16] https://www.aptoide.com/en/home.

```
void crashOnInit () {
  gDvm.methDalvikDdmcServer_dispatch = NULL;
}

// Vtable structure. Just to make messing around with it more intuitive
struct VT_JdwpAdbState {
unsigned long x;
unsigned long y;
void * JdwpSocketState_destructor;
void * _JdwpSocketState_destructor;
void * Accept;
void * showmanyc;
void * ShutDown;
void * ProcessIncoming;
};

void tamperJDWPDebugger() {
  void* lib = dlopen("libart.so", RTLD_NOW);
  if (lib == NULL) {
    log("Error loading libart.so");
    dlerror();
  }
  else{
    struct VT_JdwpAdbState *vtable = ( struct VT_JdwpAdbState *);
    dlsym(lib, "_ZTVN3art4JDWP12JdwpAdbStateE");
    if (vtable == 0) {
      log("Couldn't resolve symbol '_ZTVN3art4JDWP12JdwpAdbStateE'.\n");
    }
    else {
      log("Vtable for JdwpAdbState at: %08x\n", vtable);
      // Let the fun begin!
      unsigned long pagesize = sysconf(_SC_PAGE_SIZE);
      unsigned long page = (unsigned long)vtable & ~(pagesize-1);
      mprotect((void *)page, pagesize, PROT_READ | PROT_WRITE);
      vtable->ProcessIncoming = vtable->ShutDown;
      // Reset permissions & flush cache
      mprotect((void *)page, pagesize, PROT_READ);
    }
  }
}
```

**Fig. A7.** Native implementation of *Altering Debugging Memory Structure* protection.

```
public static final String originalSignature = "478yYkKAQF+KST8y4ATKvHkYibo=";

public static int checkAppSignature(Context context) throws Exception {
  PackageInfo packageInfo = context.getPackageManager().getPackageInfo(
    context.getPackageName(), PackageManager.GET_SIGNATURES);
  for (Signature signature : packageInfo.signatures) {
    byte[] signatureBytes = signature.toByteArray();
    MessageDigest md = MessageDigest.getInstance("SHA");
    md.update(signature.toByteArray());
    final String currentSignature = Base64.encodeToString(md.digest(), Base64.DEFAULT);
    if (originalSignature.equals(currentSignature)){
      return true;
    }
  }
  return false;
}
```

**Fig. A8.** Java implementation of *Signature Checking* protection.

- *SafetyNet attestation*: An app can invoke SafetyNet to verify the integrity of the smartphone in which it is running. SafetyNet can provide information on alterations such as rooting or bootloader unlocking. Usually, attackers exploit these features to install dynamic analysis frameworks. Furthermore, SafetyNet can also provide information about the app that invoked the service, like the signature. This information can be used to perform integrity checks on the app itself. The example implementation for this protection is rather long and we do not report it here. Therefore, we leave the reference for further insights.[17] Instead, we report a sample output JSON in Fig. A12 (page 31). The *ctsProfileMatch* and *basicIntegrity* fields provide spot checks for device integrity. The *apkPackageName* and the *apkDigestSha256* fields give indications on the integrity of the package of the app. The *apkCertificateDigestSha256* field contains information on the integrity of the certificate of the app.

[17] https://github.com/googlesamples/android-play-safetynet.

```
jbyteArray getSignatureFromNative () {
  NSV_LOGI("pathHelperGetPath starts\n");
  char *path = pathHelperGetPath();
  NSV_LOGI("pathHelperGetPath finishes\n");
  if (!path) {
    return NULL;
  }
  NSV_LOGI("pathHelperGetPath result[%s]\n", path);
  NSV_LOGI("unzipHelperGetCertificateDetails starts\n");
  size_t len_in = 0;
  size_t len_out = 0;
  unsigned char *content = unzipHelperGetCertificateDetails(path, &len_in);
  NSV_LOGI("unzipHelperGetCertificateDetails finishes\n");
  if (!content) {
    free(path);
    return NULL;
  }
  NSV_LOGI("pkcs7HelperGetSignature starts\n");
  unsigned char *res = pkcs7HelperGetSignature(content, len_in, &len_out);
  NSV_LOGI("pkcs7HelperGetSignature finishes\n");
  jbyteArray jbArray = NULL;
  if (NULL != res || len_out != 0) {
    jbArray = (*env)->NewByteArray(env, len_out);
    (*env)->SetByteArrayRegion(env, jbArray, 0, len_out, (jbyte *) res);
  }
  free(content);
  free(path);
  pkcs7HelperFree();
  return jbArray;
}
```

**Fig. A9.** `Native` implementation of *Signature Checking* protection.

```
public static final String originalCRC = "9Guy6DJ6+gh5uSJ5sJK67=";

public boolean checkCRC (long storedCRC) {
  ZipFile zf = new ZipFile(Main.MyContext.getPackageCodePath());
  ZipEntry ze = zf.getEntry(``classes.dex'');
  return (ze.getCRC() == originalCRC);
}
```

**Fig. A10.** `Java` implementation of *Code Integrity Checking* protection.

```
private static final String playStoreAppPackageName = "com.android.vending";

public static boolean verifyInstaller(final Context context) {
  final String installer = context.getPackageManager().
    getInstallerPackageName(context.getPackageName());
  return playStoreAppPackageName.equals(installer);
}
```

**Fig. A11.** `Java` implementation of *Installer Verification* protection.

```
{
  'nonce'                       : 'R2Rra24fVm5xa2Mg',
  'timestampMs'                 : 9860437986543,
  'apkPackageName'              : 'com.package.name.of.requesting.app',
  'apkCertificateDigestSha256': ['base64 SHA-256 hash of the certificate used to sign the APK'],
  'apkDigestSha256'             : ['base64 SHA-256 hash of the APK'],
  'ctsProfileMatch'             : true,
  'basicIntegrity'              : true
}
```

**Fig. A12.** Sample output of the invocation of the `SafetyNet` service.

## Appendix B. Protection atoms

This appendix contains the protection atoms extracted from the protections. For each protection, we report the protection atoms in a table. We divide the `Java` protection atoms into sets of classes, methods, attributes and strings and the `Native` protection atoms into sets of imported symbols and strings. Note that not every protection has both `Java` and `Native` protection atoms. Note also that we extended the protection atoms with code with similar functionalities of the example implementation.

### *B1. AD protections*

- *Emulator detection* - `Java` protection atoms in Table B.1, page 32, `Native` protection atoms in Fig. B.2, page 33
- *Dynamic analysis framework detection* - `Java` protection atoms in Table B.3, page 33, `Native` protection atoms in Fig. B.4, page 34
- *Debugger detection* - `Java` protection atoms in Table B.5, page 34, `Native` protection atoms in Fig. B.6, page 34
- *Debuggable status detection* - `Java` protection atoms in Table B.7, page 34, `Native` protection atoms in Fig. B.8, page 34

**Table B1**
Protection Atomss for the *Emulator Detection* protection at `Java` level.

| | | | | |
|---|---|---|---|---|
| | c1 | `java/lang/Class` | c2 | `java/lang/reflect/Method` |
| | c3 | `android/os/Build` | c4 | `android/os/Process` |
| | c5 | `java/lang/Runtime` | c6 | `java/lang/System` |
| Classes | c7 | `android/app/ActivityManager` | | |
| | m1 | `android/app/ActivityManager.isUserAMonkey` | m2 | `java/lang/Class.forName` |
| | m3 | `java/lang/Class.getMethod` | m4 | `java/lang/reflect/Method.invoke` |
| Methods | m5 | `java/lang/Runtime.getRuntime` | m6 | `java/lang/Runtime.exec` |
| | a1 | `android/os/Build.HARDWARE` | a2 | `android/os/Build.BOARD` |
| | a3 | `android/os/Build.BRAND` | a4 | `android/os/Build.DEVICE` |
| | a5 | `android/os/Build.FINGERPRINT` | a6 | `android/os/Build.MODEL` |
| Attributes | a7 | `android/os/Build.MANUFACTURER` | a8 | `android/os/Build.PRODUCT` |
| | s1 | *android.os.SystemProperties* | s2 | *getprop* |
| | s3 | *ro.hardware* | s4 | *ro.boot.hardware* |
| | s5 | *ro.kernel.androidboot.hardware* | s6 | *ro.product.board* |
| | s7 | *ro.board.platform* | s8 | *ro.product.brand* |
| | s9 | *ro.product.device* | s10 | *ro.cm.device* |
| | s11 | *ro.bootimage.build.fingerprint* | s12 | *ro.build.fingerprint* |
| | s13 | *ro.product.manufacturer* | s14 | *ro.product.model* |
| | s15 | *goldfish* | s16 | *ranchu* |
| | s17 | *vbox86* | s18 | *ttVM_x86* |
| | s19 | *unknown* | s20 | *generic* |
| | s21 | *nox* | s22 | *FRF91* |
| Strings | s23 | *google_sdk* | s24 | *generic_x86* |
| | s25 | *generic_x86_64* | s26 | *Andy* |
| | s27 | *Droid4X* | s28 | *vbox* |
| | s29 | *Genymotion* | s30 | *ro.kernel.qemu* |
| | s31 | *qemud* | s32 | *qemu.sf.lcd_density* |
| | s33 | *qemu.hw.mainkeys* | s34 | *qemu.sf.fake_camera* |
| | s35 | */dev/socket/qemud* | s36 | */dev/qemu_pipe* |
| | s37 | */system/lib/libc_malloc_debug_qemu.so* | s38 | */sys/qemu_trace* |
| | s39 | */system/bin/qemu-props* | s40 | */dev/socket/genyd* |
| | s41 | */dev/socket/baseband_genyd* | s42 | *ro.kernel.android.qemud* |
| | s43 | *ro.kernel.qemu.gles* | s44 | *init.svc.qemud* |
| | s45 | *init.goldfish.rc* | s46 | *init.svc.qemu-props* |

**Table B2**
Protection Atomss for the *Emulator Detection* protection at `Native` level.

| Imported symbols | | | | |
|---|---|---|---|---|
| | s1 | *ro.hardware* | s2 | *ro.boot.hardware* |
| | s3 | *ro.kernel.androidboot.hardware* | s4 | *ro.product.board* |
| | s5 | *ro.board.platform* | s6 | *ro.product.brand* |
| | s7 | *ro.product.device* | s8 | *ro.cm.device* |
| | s9 | *ro.bootimage.build.fingerprint* | s10 | *ro.build.fingerprint* |
| | s11 | *ro.product.manufacturer* | s12 | *ro.product.model* |
| | s13 | *goldfish* | s14 | *ranchu* |
| | s15 | *vbox86* | s16 | *ttVM_x86* |
| | s17 | *unknown* | s18 | *generic* |
| | s19 | *nox* | s20 | *FRF91* |
| | s21 | *google_sdk* | s22 | *generic_x86* |
| Strings | s23 | *generic_x86_64* | s24 | *Andy* |
| | s25 | *Droid4X* | s26 | *vbox* |
| | s27 | *Genymotion* | s28 | *ro.kernel.qemu* |
| | s29 | *qemud* | s30 | *qemu.sf.lcd_density* |
| | s31 | *qemu.hw.mainkeys* | s32 | *qemu.sf.fake_camera* |
| | s33 | */dev/socket/qemud* | s34 | */dev/qemu_pipe* |
| | s35 | */system/lib/libc_malloc_debug_qemu.so* | s36 | */sys/qemu_trace* |
| | s37 | */system/bin/qemu-props* | s38 | */dev/socket/genyd* |
| | s39 | */dev/socket/baseband_genyd* | s40 | *ro.kernel.android.qemud* |
| | s41 | *ro.kernel.qemu.gles* | s42 | *init.svc.qemud* |
| | s45 | *init.goldfish.rc* | s44 | *init.svc.qemu-props* |

- *Altering debugger memory structure* - `Native` protection atoms in Table B.9, page 35
- *Signature checking* - `Java` protection atoms in Table B.10, page 34, `Native` protection atoms in Table B.11, page 36
- *Code integrity checking* - `Java` protection atoms in Table B.12, page 36
- *Installer verification* - `Java` protection atoms in Table B.13, page 36
- *SafetyNet attestation* - `Java` protection atoms in Table B.14, page 33

**Table B3**
Protection Atomss for the *Dynamic Analysis Framework Detection* protection at `Java` level.

| | | | | |
|---|---|---|---|---|
| Classes | c1 | `dalvik/system/DexFile` | c2 | `java/lang/StackTraceElement` |
| | c3 | `android/app/ActivityManager$ RunningServiceInfo` | | |
| | c4 | `android/app/ActivityManager` | c5 | `android/content/pm/ApplicationInfo` |
| | c6 | `java/util/Enumeration` | c7 | `java/lang/reflect/Modifier` |
| Methods | m1 | `java/lang/StackTraceElement.getClassName` | | |
| | m2 | `java/lang/StackTraceElement.getMethodName` | | |
| | m3 | `android/app/ActivityManager.getRunningServices` | | |
| | m4 | `android/content/Context.getPackageCodePath` | | |
| | m5 | `java/lang/reflect/Modifier.isNative` | m6 | `dalvik/system/DexFile.entries` |
| | m7 | `java/util/Enumeration.hasMoreElements` | m8 | `java/util/Enumeration.nextElement` |
| Attributes | a1 | `android/content/pm/ApplicationInfo.sourceDir` | | |
| | a2 | `android/app/ActivityManager$RunningServiceInfo.process` | | |
| | a3 | `android/content/pm/ApplicationInfo.processName` | | |
| Strings | s1 | *com.saurik.substrate* | s2 | *com.saurik.substrate.MS$2* |
| | s3 | *de.robv.android.xposed.XposedBridge* | s4 | *XposedBridge.jar* |
| | s5 | *xposed* | s6 | *fridaserver* |
| | s7 | *LIBFRIDA* | s8 | *frida* |
| | s9 | *frida-gadget* | s10 | *frida-agent* |
| | s11 | */proc/self/maps* | s12 | *classes.dex* |
| | s13 | *classes2.dex* | s14 | *classes3.dex* |
| | s15 | *classes4.dex* | s16 | *classes5.dex* |

**Table B4**
Protection Atomss for the *Dynamic Analysis Framework Detection* protection at `Native` level.

| Imported symbols | | | | |
|---|---|---|---|---|
| Strings | s1 | *com.saurik.substrate* | s2 | *com.saurik.substrate.MS$2* |
| | s3 | *de.robv.android.xposed.XposedBridge* | s4 | *XposedBridge.jar* |
| | s5 | *xposed* | s6 | *fridaserver* |
| | s7 | *LIBFRIDA* | s8 | *frida* |
| | s9 | *frida-gadget* | s10 | *frida-agent* |
| | s11 | *127.0.0.1* | s12 | *REJECT* |
| | s13 | */proc/self/maps* | | |

**Table B5**
Protection Atomss for the *Debugger Detection* protection at `Java` level.

| Classes | c1 | `android/os/Debug` | c2 | `android/app/ActivityManager` |
|---|---|---|---|---|
| Methods | m1 | `android/os/Debug.isDebuggerConnected` | | |
| | m2 | `android/os/Debug.waitingForDebugger` | | |
| | m3 | `android/app/ActivityManager.isRunningInTestHarness` | | |
| Attributes | | | | |
| Strings | s1 | *TracerPid* | s2 | */proc/self/status* |
| | s3 | */proc/* | s4 | */status* |
| | s5 | *pid* | | |

**Table B6**
Protection Atomss for the *Debugger Detection* Protection at `Native` level.

| | | | | |
|---|---|---|---|---|
| Imported Symbols | 1i | `fork` | 2i | `getppid` |
| | 3i | `ptrace` | 4i | `waitpid` |
| | 5i | `pthread_create` | 6i | `pthread_exit` |
| | 7i | `WIFSTOPPED` | 8i | `pthread_t` |
| Strings | s1 | *TracerPid* | s2 | */proc/self/status* |
| | s3 | */proc/* | s4 | */status* |
| | s5 | *pid* | | |

**Table B7**
Protection Atomss for the *Debuggable Status Detection* protection at `Java` level.

| | | | | |
|---|---|---|---|---|
| Classes | c1 | `android/content/Context` | c2 | `android/content/pm/ApplicationInfo` |
| | c3 | `android/content/pm/ApplicationInfo` | c4 | `android/os/Process` |
| | c5 | `substituteWithTheApplicationPackage/BuildConfig` | | |
| Methods | m1 | `android/content/Context.getApplicationInfo` | m2 | `java/lang/Runtime.getRuntime` |
| | m3 | `java/lang/Runtime.exec` | | |
| Attributes | a1 | `android/content/pm/ApplicationInfo.flags` | | |
| | a2 | `android/content/pm/ApplicationInfo.FLAG_DEBUGGABLE` | | |
| | a3 | `substituteWithTheApplicationPackage/BuildConfig.DEBUG` | | |
| Strings | s1 | *ro.debuggable* | s2 | *getprop* |
| | s3 | *android.os.SystemProperties* | | |

**Table B8**
Protection Atomss for the *Debuggable Status Detection* protection at `Native` level.

| Imported symbols | | |
|---|---|---|
| Strings | s1 | *ro.debuggable* |

**Table B9**
Protection Atomss for the *Altering Debugger Memory Structure* Protection at `Native` Level.

| | | | | |
|---|---|---|---|---|
| Imported Symbols | 1i | gDvm | | |
| Strings | s1 | *libart.so* | s2 | *_ZTVNa3rt4JDWP12JdwpAdbStateE* |

**Table B10**
Protection Atomss for the *Signature Checking* protection at `Java` level.

| | | | | |
|---|---|---|---|---|
| Classes | c1 | `java/security/MessageDigest` | c2 | `android/content/pm/PackageInfo` |
| | c3 | `android/content/pm/Signature` | c4 | `android/content/pm/PackageManager` |
| | c5 | `android/content/Context` | c6 | `android/content/pm/VersionedPackage` |
| | c7 | `android/content/pm/SigningInfo` | | |
| Methods | m1 | `java/security/MessageDigest.getInstance` | | |
| | m2 | `java/security/MessageDigest.update` | | |
| | m3 | `java/security/MessageDigest.digest` | | |
| | m4 | `android/content/pm/PackageManager.getPackageInfo` | | |
| | m5 | `android/content/pm/Signature.toByteArray` | | |
| | m6 | `android/content/Context.getPackageManager` | | |
| | m7 | `android/content/Context.getPackageName` | | |
| | m8 | `android/content/pm/SigningInfo.getApkContentsSigners` | | |
| | m9 | `android/content/pm/SigningInfo.getSigningCertificateHistory` | | |
| Attributes | a1 | `android/content/pm/PackageManager.GET_SIGNATURES` | | |
| | a2 | `android/content/pm/PackageManager.GET_SIGNING_CERTIFICATES` | | |
| | a3 | *android/content/pm/PackageInfo.signatures* | | |
| Strings | s1 | *MD2* | s2 | *MD5* |
| | s3 | *SHA* | s4 | *SHA-1* |
| | s5 | *SHA-224* | s6 | *SHA-256* |
| | s7 | *SHA-384* | s8 | *SHA-512* |
| | s9 | *No package found for authority:* | s10 | *Found content provider* |
| | s11 | *, but package was not* | | |

**Table B11**
Protection Atomss for the *Signature Checking* protection at `Native` level.

| Imported Symbols | | | | |
|---|---|---|---|---|
| Strings | s1 | *META-INF/* | s2 | *.RSA* |
| | s3 | *.DSA* | s4 | *.EC* |
| | s5 | */proc/self/cmdline* | s6 | */proc/self/maps* |
| | s7 | *tbsCertificate* | s8 | *version* |
| | s9 | *serialNumber* | s10 | *signature* |
| | s11 | *issuer* | s12 | *validity* |
| | s13 | *subject* | s14 | *subjectPublicKeyInfo* |
| | s15 | *issuerUniqueID-[optional]* | s16 | *subjectUniqueID-[optional]* |
| | s17 | *extensions-[optional]* | s18 | *signatureAlgorithm* |
| | s19 | *signatureValue* | s20 | *version* |
| | s21 | *issuerAndSerialNumber* | s22 | *digestAlgorithmId* |
| | s23 | *authenticatedAttributes-[optional]* | s24 | *digestEncryptionAlgorithmId* |
| | s25 | *encryptedDigest* | s26 | *unauthenticatedAttributes-[optional]* |
| | s27 | *DigestAlgorithms* | s28 | *contentInfo* |
| | s29 | *crls-[optional]* | s30 | *signerInfos* |
| | s31 | *signerInfo* | | |

**Table B12**
Protection Atomss for the *Code Integrity Checking* protection at Java level.

| | | | | |
|---|---|---|---|---|
| | c1 | java/util/zip/ZipFile | c2 | java/util/zip/ZipEntry |
| | c3 | java/util/jar/JarFile | c4 | java/util/jar/JarEntry |
| Classes | c5 | java/util/zip/Adler32 | c6 | java/util/zip/CRC32 |
| | c7 | android/content/Context | c8 | android/content/pm/ApplicationInfo |
| | m1 | android/content/Context.getPackageCodePath | | |
| | m2 | java/util/jar/JarEntry.getCrc | m3 | java/util/zip/ZipEntry.getCrc |
| | m4 | java/util/zip/Adler32.update | m5 | java/util/zip/CRC32.update |
| Methods | m6 | java/util/zip/Adler32.getValue | m7 | java/util/zip/CRC32.getValue |
| | m8 | java/util/zip/ZipFile.getEntry | m9 | java/util/zip/ZipFile.entries |
| | m10 | java/util/jar/JarFile.getEntry | m11 | java/util/jar/JarFile.entries |
| | m12 | java/util/jar/JarFile.getJarEntry | m13 | android/content/Context.getString |
| Attributes | a3 | android/content/pm/ApplicationInfo.sourceDir | | |
| | s1 | *classes.dex* | s2 | *classes2.dex* |
| Strings | s3 | *classes3.dex* | s4 | *classes4.dex* |
| | s5 | *classes5.dex* | s6 | *MultiDexExtractor.load(* |

**Table B13**
Protection Atomss for the *Installer Verification* protection at Java level.

| | | | | |
|---|---|---|---|---|
| | c1 | android/content/pm/PackageInfo | c2 | android/content/pm/PackageManager |
| Classes | c3 | android/content/Context | | |
| | m1 | android/content/pm/PackageManager.getInstallerPackageName | | |
| | m2 | android/content/Context.getPackageManager | | |
| Methods | m3 | android/content/Context.getPackageName | | |
| | m4 | android/content/pm/PackageManager.getPackageInfo | | |
| | a1 | android/content/pm/PackageInfo.packageName | | |
| Attributes | a2 | android/content/pm/PackageInfo.versionCode | | |
| | a3 | android/content/pm/PackageInfo.versionName | | |
| | s1 | *com.android.vending* | s2 | *com.amazon.venezia* |
| | s3 | *com.sec.android.app.samsungapps* | s4 | *cm.aptoide.pt* |
| Strings | s5 | *org.fdroid.fdroid* | s6 | *com.uptodown* |
| | s7 | *com.uptodown.lite* | s8 | *com.slideme.sam.manager* |

**Table B14**
Protection Atomss for the *SafetyNet Attestation* protection at Java level.

| | | | | |
|---|---|---|---|---|
| | c1 | com/google/android/gms/safetynet/SafetyNet | | |
| | c2 | com/google/android/gms/safetynet/SafetyNetClient | | |
| Classes | c3 | com/google/android/gms/safetynet/SafetyNetApi | | |
| | c4 | com/google/android/gms/safetynet/SafetyNetApi$AttestationResponse | | |
| | m1 | com/google/android/gms/safetynet/SafetyNet.getClient | | |
| Methods | m2 | com/google/android/gms/safetynet/SafetyNetClient.attest | | |
| | m3 | com/google/android/gms/safetynet/SafetyNetApi$AttestationResponse.getJwsResult | | |
| Attributes | | | | |
| | s1 | *basicIntegrity* | s2 | *ctsProfileMatch* |
| | s3 | *apkDigestSha256* | s4 | *apkCertificateDigestSha256* |
| Strings | s5 | *apkPackageName* | s6 | *timestampMs* |
| | s7 | *nonce* | | |

## Appendix C. Fingerprints derived from the protections

This appendix presents the fingerprints of the protections. We singularly present the protection atoms relevant to each protection and then the fingerprints. Note that in the fingerprints there are not protection atoms related to classes. We already include them in the detection of the methods and the attributes. In practice, detecting a method or an attribute of a class implies the presence of the class itself.

### C1. AD protections

- *Emulator detection* - `Java` fingerprint in Table C.1, page 38, `Native` fingerprint in Table C.2, page 38
- *Dynamic Analysis Framework Detection* - `Java` fingerprint in Table C.3, page 38, `Native` fingerprint in Table C.4, page 39
- *Debugger Detection* - `Java` fingerprint in Table C.5, page 39, `Native` fingerprint in Table C.6, page 39
- *Debuggable Status Detection* - `Java` fingerprint in Table C.7, page 39, `Native` fingerprint in Figure C.8, page 39
- *Altering Debugger Memory Structure* - `Native` fingerprint in Table C.9, page 39

**Table C1**
Fingerprint for the *Emulator Detection* protection at `Java` level.

| Condition | Protection Atomss | Description |
|---|---|---|
| $A$ | `s1-2` | "*Android.os.SystemProperties*" or "*getprop*" |
| $B$ | `s3-14` | Strings for getting smartphone properties |
| $C$ | `a1-8` | `android.os.Build` attributes |
| $D$ | `s15-18 s21-29` | string for comparison of properties |
| $E$ | `s30-47` | emulator related strings |
| $F$ | `m1` | the `isUserAMonkey` method |
| | $(((A \wedge B) \vee C) \wedge D) \vee E \vee F$ | |

**Table C2**
Fingerprint for the *Emulator Detection* protection at `Native` level.

| Condition | Protection Atomss | Description |
|---|---|---|
| $A$ | `s1-12` | strings for getting properties |
| $B$ | `s13-16 s19-27` | string for comparison of properties |
| $C$ | `s28-45` | emulator related strings |
| | $((A \wedge B) \vee C)$ | |

**Table C3**
Fingerprint for the *Dynamic Analysis Framework Detection* protection at `Java` level.

| Condition | Protection Atomss | Description |
|---|---|---|
| $A$ | `s1-19` | At least one of strings related to the frameworks |
| $B$ | `m1-2` | At least one of the methods for handling a thrown exception |
| $C$ | `m3` | The method for getting the running services |
| $D$ | `s12-16` | At least one of the strings for the *.dex* file |
| $E$ | `a1` | `ApplicationInfo.sourceDir` for the path of the APK |
| $F$ | `m4` | method for getting the path of the APK |
| $G$ | `m5` | check if a method is native thus hooked |
| $H$ | `s11` | "*/proc/self/maps*" string |
| $I$ | `a2-3` | get the name of the process |
| | | $A \vee ((E \vee F) \wedge D \wedge G)$ |

**Table C4**
Fingerprint for the *Dynamic Analysis Framework Detection* Protection at `Native` level.

| Condition | Protection Atomss | Description |
|---|---|---|
| $A$ | `s1-10` | At least one of strings related to the frameworks |
| | | $A$ |

**Table C5**
Fingerprint for the *Debugger Detection* Protection at `Java` level.

| Condition | Protection Atomss | Description |
|---|---|---|
| $A$ | `m1-3` | Methods related to the presence of a debugger |
| $B$ | `s1` | "*TracerPid*" string |
| $C$ | `s2` | "*/proc/self/status*" string |
| $D$ | `s3-4` | both "*/proc/*" + "*/status*" strings |
| | | $A \vee (B \wedge (C \vee D))$ |

**Table C6**
Fingerprint for the *Debugger Detection* Protection at `Native` level.

| Condition | Protection Atomss | Description |
|---|---|---|
| *A* | `i1-4` | `fork`, `getpid`, `ptrace` or `waitpid` symbols |
| *B* | `s1` | "*TracerPid*" string |
| *C* | `s2` | "*/proc/self/status*" string |
| *D* | `s3-4` | both "*/proc/*" + "*/status*" strings |
| | $A \vee (B \wedge (C \vee D))$ | |

**Table C7**
Fingerprint for the *Debuggable Status Detection* Protection at `Java` level.

| Condition | Protection Atomss | Description |
|---|---|---|
| *A* | `s1` | "*ro.debuggagle*" string for the system property |
| *B* | `s2-3` | "*Android.os.SystemProperties*" or "*getProp*" strings |
| *C* | `a2-3` | At least one of the attributes related to a debuggable status |
| | $(A \wedge B) \vee (C)$ | |

**Table C8**
Fingerprint for the *Debuggable Status Detection* Protection at `Native` Level.

| Condition | Protection Atomss | Description |
|---|---|---|
| *A* | `s1` | "*ro.debuggagle*" string |
| | $A$ | |

**Table C9**
Fingerprint for the *Altering Debugger Memory Structure* Protection at `Native` Level.

| Condition | Protection Atomss | Description |
|---|---|---|
| *A* | `s1-2` | both the strings extracted from the ART protection |
| *B* | `i1` | the `gDvm` symbol for DALVIK |
| | $A \vee B$ | |

**Table C10**
Fingerprint for the *Signature Checking* Protection at `Java` Level.

| Condition | Protection Atomss | Description |
|---|---|---|
| *A* | `s1-8` | At least one of strings for digest algorithm |
| *B* | `m1-3` | All of methods for digest |
| *C* | `m8-9` | At least one of methods for signatures |
| *D* | `a1-3` | At least one of attribute for signatures |
| | $A \wedge B \wedge (C \vee D)$ | |

**Table C11**
Fingerprint for the *Signature Checking* Protection at `Native` Level.

| Condition | Protection Atomss | Description |
|---|---|---|
| *A* | `s1-6` | All of the string for getting the certificate |
| | | $A$ |

### *C2. AD Protections*

- *Signature Checking* - `Java` fingerprint in Table C.10, page 40, `Native` fingerprint in Table C.11, page 40
- *Code Integrity Checking* - `Java` fingerprint in Table C.12, page 40
- *Installer Verification* - `Java` fingerprint in Table C.13, page 40
- *SafetyNet Attestation* - `Java` fingerprint in Table C.14, page 41

**Table C12**
Fingerprint for the *Code Integrity Checking* Protection at `Java` level.

| Condition | Protection Atomss | Description |
|---|---|---|
| *A* | `s1-5` | *classes.dex* strings |
| *B* | `m1` | Method for getting the package code path |
| *C* | `a1` | Attribute to get the package code path |
| *D* | `m2-3` | At least one of the methods for the CRC |
| *E* | `m4-7` | At least one of the methods for the CRC |
| | $A \wedge (B \vee C) \wedge (D \mid \vee E)$ | |

**Table C13**
Fingerprint for the *Installer Verification* Protection at `Java` Level.

| Condition | Protection Atomss | Description |
|---|---|---|
| *A* | `s1-8` | At least one of the stores names |
| *B* | `m1` | method `getInstallerPackageName` |
| | $A \wedge B$ | |

**Table C14**
Fingerprint for the *SafetyNet Attestation* Protection at `Java` Level.

| Condition | Protection Atomss | Description |
|---|---|---|
| *A* | `m1-3` | At least one of the methods |
| *B* | `c1-4` | At least one of the classes |
| | $A \wedge B$ | |

## Appendix D. List of libraries filtered

The list of package names of third-party libraries in Table D1 is by no means complete. Indeed, future work consists also of enriching this collection. The "*" character is the wildcard character.

**Table D1**
Third-party libraries filtered.

| | | | |
|---|---|---|---|
| android.* | androidx.* | butterknife.* | com.android.* |
| com.adcolony.* | com.adjust.* | com.crittercism.* | com.readystatesoftware.* |
| com.appsflyer.* | com.networkbench.* | com.dropbox.* | com.braintreepayments.* |
| com.airbnb.lottie.* | com.jakewharton.* | com.rateus.* | com.twitter.* |
| com.comscore.* | com.my.target.* | com.startapp.* | com.mobvista.* |
| com.facebook.* | com.monet.* | com.samsung.* | com.kochava.* |
| com.baidu.* | com.tune.* | com.amazon.* | com.moat.* |
| com.inmobi.* | com.flurry.* | com.tencent.* | com.paypal.* |
| com.distil.* | com.google.* | com.zendesk.* | com.bugsnag.* |
| com.applovin.* | com.squareup.* | com.foursquare.* | com.mixpanel.* |
| com.getkeepsafe.* | com.qihoo360.* | com.anjlab.* | com.scottyab.* |
| com.unity3d.* | com.zopim.* | com.learnium.* | com.crashlytics.* |
| com.stripe.* | com.umeng.* | cn.jiguang.* | dalvik.* |
| dagger.* | de.blinkt.openvpn.* | java.* | javax.* |
| io.fabric.* | io.agora.* | io.sentry.* | io.intercom.* |
| io.branch.* | io.reactivex.* | io.realm.* | net.hockeyapp.* |
| net.openid.* | org.acra.* | org.spongycastle.* | org.xbill.* |
| okio.gzipsing.* | org.apache.* | org.chromium.* | org.conscrypt.* |
| org.mozilla.* | org.sufficientlysecure.* | org.godotengine.* | org.webrtc.* |
| okhttp3.* | org.greenrobot.* | org.robolectric.* | org.parceler.* |
| retrofit2.* | kotlin.* | kotlinx.* | |

## References

[1] Li L, Bissyand T, Klein J. Rebooting research on detecting repackaged android apps: Literature review and benchmark. IEEE Trans Softw Eng 2019;PP. doi:10.1109/TSE.2019.2901679. 1–1.

[2] Sommerlad J.. Spotify cracks down on premium pirates streaming for free. 2018. URL independent.co.uk/life-style/gadgets-and-tech/news/spotify-premium-piracy-crackdown-apps-bypass-restrictions-accounts-deactivated-music-streaming-a8241936.html.

[3] ustwo games. Twitter status. 2015. URL twitter.com/ustwogames/status/552136427904184320.

[4] Ceccato M, Tonella P, Basile C, Falcarin P, Torchiano M, Coppens B, et al. Understanding the behaviour of hackers while performing attack tasks in a professional setting and in a public challenge. Empir Softw Eng 2019;24(1):240–86. doi:10.1007/s10664-018-9625-6.

[5] Piao Y, Jung J, Hyun Yi J. Server-based code obfuscation scheme for apk tamper detection. Secur Commun Netw 2014;9. doi:10.1002/sec.936.

[6] Zhou W, Wang Z, Zhou Y, Jiang X. Divilar: diversifying intermediate language for anti-repackaging on android platform; 2014. p. 199–210. doi:10.1145/2557547.2557558.

[7] Vidas T, Christin N. Evading android runtime analysis via sandbox detection. In: Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security. New York, NY, USA: ACM; 2014. p. 447–58. ISBN 978-1-4503-2800-5. doi:10.1145/2590296.2590325.

[8] Developers G.. Android studio documentation and guidelines. 2018a. URL developer.android.com/docs/.

[9] Foundation T.O.. Owasp mobile security testing guide. 2018. URL https://mobile-security.gitbook.io/mobile-security-testing-guide/.

[10] Balebako R, Marsh A, Lin J, Hong J, Cranor L. The privacy and security behaviors of smartphone app developers; 2014. ISBN 1-891562-37-1. doi:10.14722/usec.2014.23006.

[11] Alexander-Bown S.. Android security: adding tampering detection to your app. URL airpair.com/android/posts/adding-tampering-detection-to-your-android-app.

[12] Kozhevin D.. Native signature verification for android with example. 2018. URL github.com/DimaKoz/stunning-signature.

[13] Fenton C.. Android emulator detection. 2016. URL https://github.com/CalebFenton/AndroidEmulatorDetect.

[14] Enck W, Gilbert P, Han S, Tendulkar V, Chun B-G, Cox LP, et al. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Trans Comput Syst 2014;32(2) 5:1–5:29. doi:10.1145/2619091.

[15] Zhou W, Zhou Y, Jiang X, Ning P. Detecting repackaged smartphone applications in third-party android marketplaces. In: Proceedings of the Second ACM Conference on Data and Application Security and Privacy. New York, NY, USA: ACM; 2012. p. 317–26. ISBN 978-1-4503-1091-8. doi:10.1145/2133601.2133640.

[16] Developers G.. Protect against security threats with safetynet. 2018b. URL developer.android.com/training/safetynet.

[17] Developers G.. Security tips. 2018c. URL https://developer.android.com/training/articles/security-tips/.

[18] Li L, Bissyand TF, Octeau D, Klein J. DroidRA: taming reflection to support whole-program analysis of Android apps. In: Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016. Saarbr&252;cken, Germany: ACM Press; 2016. p. 318–29. ISBN 978-1-4503-4390-9. doi:10.1145/2931037.2931044.

[19] Bruneton E, Lenglet R, Coupaye T. Asm: a code manipulation tool to implement adaptable systems. In Adaptable and extensible component systems; 2002.

[20] Li L, Gao J, Hurier M, Kong P, Bissyandé T, Bartel A, et al. Androzoo++: collecting millions of android apps and their metadata for the research community; 2017.

[21] Devore JL. Probability and statistics for engineering and the sciences. Duxbury Press; 7 edition; 2007.

[22] Wang Y, Rountev A. Who changed you? obfuscator identification for android; 2017. p. 154–64. doi:10.1109/MOBILESoft.2017.18.

[23] Wermke D, Huaman N, Acar Y, Reaves B, Traynor P, Fahl S. A large scale investigation of obfuscation use in google play; 2018. p. 222–35. doi:10.1145/3274694.3274726.

[24] Viticchié A, Basile C, Avancini A, Ceccato M, Abrath B, Coppens B. Reactive attestation: automatic detection and reaction to software tampering attacks. In: Proceedings of the 2016 ACM Workshop on Software PROtection. New York, NY, USA: ACM; 2016. p. 73–84. doi:10.1145/2995306.2995315.

[25] Wan J, Zulkernine M, Liem C. A dynamic app anti-debugging approach on android art runtime; 2018. p. 560–7. doi:10.1109/DASC/PiCom/DataCom/CyberSciTec.2018.00105.

[26] Abrath B, Coppens B, Volckaert S, Wijnant J, De Sutter B. Tightly-coupled self-debugging software protection. In: Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering. New York, NY, USA: ACM; 2016. 7:1–7:10 http://doi.acm.org/10.1145/3015135.3015142. ISBN 978-1-4503-4841-6.

[27] Jing Y, Zhao Z, Ahn G-J, Hu H. Morpheus: automatically generating heuristics to detect Android emulators. In: Proceedings of the 30th Annual Computer Security Applications Conference on - ACSAC '14. New Orleans, Louisiana: ACM Press; 2014. p. 216–25. ISBN 978-1-4503-3005-3. doi:10.1145/2664243.2664250.

[28] Lim K, Jeong Y, je Cho S, Park M, Han S. An android application protection scheme against dynamic reverse engineering attacks. J Wirel Mobile Netw Ubiquitous Comput Dependable Appl 2016;7(3):40–52. doi:10.22667/JOWUA.2016.09.31.040.

[29] Vasileiadis L.. Remote runtime detection of tampering and of dynamic analysis attempts for android apps. 2019. URL http://essay.utwente.nl/79200/.

[30] Ghafari M, Gadient P, Nierstrasz O. Security smells in android; 2017. p. 121–30. doi:10.1109/SCAM.2017.24.

[31] Shan Z, Neamtiu I, Samuel R. Self-hiding behavior in android apps: detection and characterization. 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE) 2018:728–39.

[32] Gao J, Kong P, Li L, Bissyande TF, Klein J. Negative results on mining Crypto-API usage rules in android apps. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR). Montreal, QC, Canada: IEEE; 2019. p. 388–98. ISBN 978-1-72813-412-3. doi:10.1109/MSR.2019.00065.

[33] Habchi S, Moha N, Rouvoy R. The rise of android code smells: who is to blame?. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR). Montreal, QC, Canada: IEEE; 2019. p. 445–56. ISBN 978-1-72813-412-3. doi:10.1109/MSR.2019.00071.

[34] Kaur R, Ning Y, Gonzalez H, Stakhanova N. Unmasking android obfuscation tools using spatial analysis; 2018. p. 1–10. doi:10.1109/PST.2018.8514207.

[35] Wang P, Bao Q, Wang L, Wang S, Chen Z, Wei T, et al. Software protection on the go: a large-scale empirical study on mobile app obfuscation. In: Proceedings of the 40th International Conference on Software Engineering. New York, NY, USA: ACM; 2018. p. 26–36. doi:10.1145/3180155.3180169.

[36] Berlato S. C.M.. Atadetector. 2018.

[37] Strazzere T.. anti-emulator. 2013.

[38] Mueller B.. The jiu-jitsu of detecting frida. 2017a. URL http://www.vantagepoint.sg/blog/90-the-jiu-jitsu-of-detecting-frida.

[39] Mueller B.. The jiu-jitsu of detecting frida. 2017b. URL http://www.vantagepoint.sg/blog/89-more-android-anti-debugging-fun.

[40] Hanna S, Huang L, Wu E, Li S, Chen C, Song D. Juxtapp: a scalable system for detecting code reuse among android applications, 7591; 2012. p. 62–81. doi:10.1007/978-3-642-37300-8_4.