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

Antonio Ruggia
antonio.ruggia@dibris.unige.it
DIBRIS - University of Genoa
Genoa, Italy

Eleonora Losiouk
eleonora.losiouk@unipd.it
University of Padua
Padua, Italy

Luca Verderame
luca.verderame@dibris.unige.it
DIBRIS - University of Genoa
Genoa, Italy

Mauro Conti
conti@math.unipd.it
University of Padua
Padua, Italy

Alessio Merlo
alessio@dibris.unige.it
DIBRIS - University of Genoa
Genoa, Italy

ABSTRACT

A growing trend in repackaging attacks exploits the Android virtualization technique, in which malicious code can run together with the victim app in a virtual container. In such a scenario, the attacker can directly build a malicious container capable of hosting the victim app instead of tampering with it, thus neglecting any anti-repackaging protection developed so far. Also, existing anti-virtualization techniques are ineffective since the malicious container can intercept - and tamper with - such controls at runtime. So far, only two solutions have been specifically designed to address virtualization-based repackaging attacks. However, their effectiveness is limited since they both rely on static taint analysis, thus not being able to evaluate code dynamically loaded at runtime.

To mitigate such a problem, in this paper we introduce MARVEL, the first methodology that allows preventing both traditional and virtualization-based repackaging attacks. MARVEL strongly relies on the virtualization technique to build a secure virtual environment where protected apps can run and be checked at runtime. To assess the viability and reliability of MARVEL, we implemented it in a tool, i.e., MARVELoid, that we tested by protecting 4000 apps with 24 different configurations of the protection parameters (i.e., 96k protection combinations). MARVELoid was able to inject the protection into 97.3% of the cases, with a processing time of 98 seconds per app on average. Moreover, we evaluated the runtime overhead on 45 apps, showing that the introduced protection has a negligible impact in terms of average CPU (<5%) and memory overhead (<0.5%).

CCS CONCEPTS

• **Security and privacy → Software reverse engineering**.

KEYWORDS

Mobile Security, Anti-repackaging techniques, Android Virtualization

ACM Reference Format:
Antonio Ruggia, Eleonora Losiouk, Luca Verderame, Mauro Conti, and Alessio Merlo. 2021. Repack Me If You Can: An Anti-Repackaging Solution Based on Android Virtualization. In *Annual Computer Security Applications Conference (ACSAC '21), December 6–10, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3485832.3488021

1 INTRODUCTION

Traditional repackaging attacks aim to lure the user into installing a malicious app which looks like a legitimate one. In particular, such attacks revolve around the following steps: reverse engineering the target app, injecting malicious code, recompiling and distributing the modified app on the app markets or through other channels. Given the similarity between the repackaged and the original apps, the user is unable to distinguish between them and can be fooled into installing the malicious app. Once the malicious app runs on the victim's phone, it can start executing the attacker's code.

A recent growing trend concerns the exploitation of the Android virtualization technique [9] to generate and distribute malicious repackaged apps in an easier way with respect to traditional repackaging attacks. Android virtualization enables an app (i.e., *container*) to create a virtual environment, separated from the Android default one, in which other apps (i.e., *plugins*) can run, while fully preserving their functionalities.

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

The popularity of the virtualization technique, confirmed by the number of downloads of the virtualization apps on the Play Store [15, 35, 36], is given by its main use case scenario, i.e., running multiple instances of the same app on a single device: if a user has two separate Telegram accounts and wants to use them simultaneously, she can have the first running in the Android environment and the second in one of the virtualized contexts. Besides legitimate uses, the virtualization technique has already paved the way for threatening attacks [9, 29, 41, 42].

With respect to traditional repackaging attacks, virtualization-based ones are easier to setup. The former require the attacker to i) decompile the Application Package (APK) of the victim app, ii) detect and, in case, remove anti-repackaging protections, iii) inject some malicious code, and iv) recompile it again. Instead, virtualization-based repackaging attacks allow the attacker to make the container execute the original victim app along with some malicious code, without requiring any modification of the original APK. The malicious code can be part of the container logic or another plugin running in the same virtual environment as the victim one. Two examples of virtualization-based malware have already been detected in the wild [3, 5]. The first one [3] targets the Twitter app and aims to support the app dual-instance execution. The user logins into his second Twitter account inside the dual-instance app, which, besides creating a virtual environment, can also hijack user input. The second malware aims to distribute an updated version of the WhatsApp app [5]. This is definitely similar to the original one, besides the additional malicious code crafted to steal users' sensitive data. The use of the virtualization technique is confirmed by the presence of a file called `WhatsApp.apk` inside the malicious container APK which is loaded at runtime as a plugin.

To defend against malicious usage of the virtualization technique, researchers have put forward some solutions [9, 19, 29, 30, 38, 41]. The majority of them are distributed as a library that plugins have to embed to detect at runtime whether they are running in a virtual environment. Limitations of such approaches are twofold: the checks they rely on can be easily bypassed [1], and they are not able to distinguish between a benign and a malicious usage of the virtualization technique. On the contrary, two solutions [30, 41] have been proposed to defend against virtualization-based repackaging attacks. Both of them rely on a static analysis approach to find, first, any usage of the virtualization technique and, then, the purpose of its usage. Being designed as static analysis tools, they are affected by well-known limitations (e.g., missing evaluation of code dynamically loaded at runtime).

Given the number of malware samples designed on top of the virtualization technique, we believe there is an urgent need to provide a reliable defense methodology that can i) prevent any repackaging attack, and ii) be adopted by any Android user, straightforwardly. To this aim, in this paper, we present MARVEL (i.e., Mobile-app Anti-Repackaging for Virtual Environments Locking), the first dynamic anti-repackaging solution based on the virtualization technique. MARVEL relies on a trusted container that creates a virtual environment, where plugins equipped with proper anti-repackaging checks run. Plugins are not able to run in the native Android OS due to the anti-repackaging checks, which rely on anti-tampering, environment evaluation and anti-debugging techniques [6, 22, 31], and are unlocked by the trusted container. To experimentally evaluate the feasibility of MARVEL, we implemented it in a tool (i.e., MARVELoid) that we leveraged to test the methodology against a dataset of 4000 apps. In particular, we protected each app with 24 different configurations of the protection parameters, ending up with 96k different protection combinations. The tool achieved the 97.3% of success rate and required - on average - only 98 seconds per app to introduce the protections. Moreover, we evaluated the protected apps at runtime, measuring the number of failures in their execution, and the overhead introduced by MARVEL in terms of CPU and required memory. The results showed that our solution introduces a limited overhead with respect to traditional virtualization techniques, i.e., an increment - on average - up to 4.7 percentage points (pp) in the CPU usage and 0.2 pp for the consumed memory.

Contributions. Contributions of this paper are as follows:

- We designed MARVEL, the first solution that prevents any repackaging attack and protects any Android user by running on mobile devices.
- We developed the MARVELoid tool to experimentally evaluate the effectiveness of MARVEL.
- We measured the efficacy of MARVEL over 4000 apps, achieving a 97.3% success rate in protecting the apps and introducing an average increment up to 4.7 percentage points (pp) in the CPU usage and 0.2 pp in consumed memory with respect to existing virtual environments.

Organization. The rest of the paper is organized as follows: Section 2 discusses the existing anti-repackaging techniques and the Android virtualization one; Section 3 summarizes previous work concerning anti-repackaging and anti-virtualization strategies; Section 4 details the threat model; design, implementation and experimental evaluation of MARVEL are illustrated in Section 5, Section 6 and Section 7, respectively. Finally, Section 8 provides some discussion and points out some future work.

2 BACKGROUND

In this section, we introduce the state-of-the-art of anti-repackaging techniques (Section 2.1) and the internal details of an Android virtual environment (Section 2.2).

2.1 Anti-Repackaging

In recent years, researchers proposed several techniques [16, 22] to discourage attackers from repackaging apps. Such techniques can be divided in two main categories, namely *repackaging detection* and *anti-repackaging* (also known as *repackaging avoidance* or *self-protection*).

The first group of techniques aims to recognize apps that have been already successfully repackaged and delivered to app stores or users' devices. Common *repackaging detection* solutions rely on app similarity, which is evaluated through a two-step process: each app is first profiled according to a set of distinctive features (e.g., call graph and control flow graph [20]); then, apps feature profiles are compared to each other and, if their similarity is lesser than a given optimal threshold (calculated according to some machine learning techniques, either supervised [37] or unsupervised [12]), the apps are considered clones, thereby indicating that one is probably the repackaged version of the other one.

The latter group (i.e., *anti-repackaging*) comprises techniques aimed at making the repackaging process more difficult for an attacker. In particular, the app developer injects detection nodes that implement *anti-tampering* (AT) controls aimed at checking some metadata of an app (e.g., the signature or the package name). If the AT controls fail at runtime, the anti-tampering mechanism commonly throws a security exception to force the app to crash [6]. Thus, an attacker has to find and disable all detection nodes to successfully repackage the app.

In addition, anti-repackaging solutions can further enhance the protection by preventing the reverse-engineering of the detection nodes. Among those, there are approaches relying on *logic bombs* [39, 40]. A logic bomb protects (i.e., hides) AT controls by encrypting their code using constant values contained in the program as encryption keys. Such keys are then removed from the code and replaced with the corresponding pre-computed hash values. Thus, the protection of the logic bomb is granted by the one-way property of cryptographic hash functions, which makes hard for the attacker to statically retrieve the original value (i.e., the decryption key) from its hash. To this aim, an attacker must leverage both static and dynamic analysis techniques to execute all bombs, intercept the decryption keys, get access to the AT controls and remove them.

Overall, an anti-repackaging technique is considered reliable if the complexity for the attacker to remove all protections in an app is more time-consuming than developing the same app from scratch.

2.2 Android Virtualization

The Android virtualization technique enables an app (i.e., *container*) to create a virtual environment where other apps (i.e., *plugins*) can run. A user can install several containers, which generate their corresponding virtual environment. In these virtual environments, the user can execute several plugins independently from the underlying Android OS and other virtual environments. DroidPlugin [34] and VirtualApp [17] are the two most well-known frameworks supporting the generation of Android virtual environments and share a common design. A virtual environment can run any Android app, both single or multiple apps at a time, as well as apps that are not installed on the device. Furthermore, it does not require any additional privileges enabled on the device (e.g., root privileges).

The virtualization technique relies on *Dynamic Code Loading* (DCL) and *Java dynamic proxy* [23]. The former enables an Android app to load external code contained in a Dalvik EXecutable (DEX) file, a Java ARchive (JAR) file, or even in the APK of another app. Such behavior allows bypassing the 64K reference limit [10] imposed by the DEX format[1]. The Java dynamic proxy allows the creation of a wrapper object that intercepts all method calls towards a specific object instance, eventually adding some functionality.

Fig. 1 shows the internal architecture of an app generating a virtual environment for a plugin. The architecture involves the following components: the Container App (i.e., container), the Plugin App (i.e., plugin), the Dynamic Proxy Module, and the ClassLoader.

First, the container extracts the bytecode of the plugin APK (i.e., the `classes.dex` file(s)) and loads them into the ClassLoader of the container. Then, it forks its process (i.e., pid_0) into a new process to host the plugin (i.e., pid_1). The container and the plugin share the same User IDentifier (UID) (i.e., uid_0). At runtime, the ClassLoader resolves classes and methods (i.e., M_i) for the plugin by returning the appropriate code (CM_i). Moreover, the container has to manage the lifecycle of all the components of the plugin. In detail, Android apps contain several components (i.e., Activities, Services, Broadcast Receivers, and Content Providers), which need

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

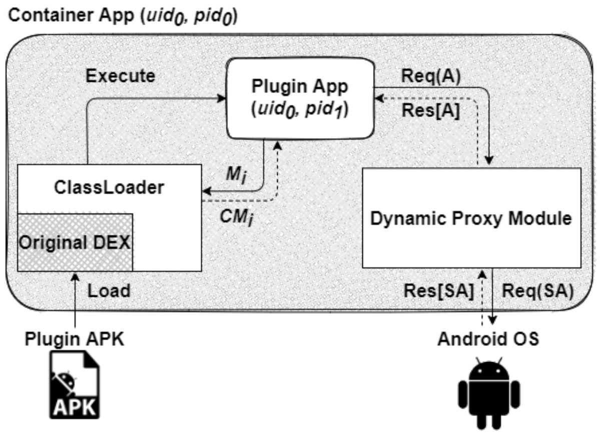


Figure 1: The internal structure of an Android virtual environment.

to be declared in the `AndroidManifest.xml` file so that the Android OS can register them at the installation time. The management of each component involves an interaction between the Android app and the Android OS: for example, to show an Activity (A) on the screen of the smartphone, an app has to send a request to open that Activity ($startActivity(A)$) to the Android OS and it has to receive a reply containing the details of the device where the app is running (e.g., the dimension of the screen). When the plugins run in a virtual environment, the Android OS receives all requests from the container. Thus, the container is expected to declare the same components as the plugins running in its virtual environment. The container first declares a generic number of *stub* components. Then, it can rely on the Dynamic Proxy Module to intercept each request (reply) going towards (coming from) the Android OS to change the name of the component dynamically. In the example of Fig. 1, the Proxy module creates a Stub Activity (SA) and sends the modified call to the Android OS ($startActivity(SA)$). Then, the result of the invocation ($Res[SA]$) is mapped back to the original component ($Res[A]$). Concerning permissions, the container requires all existing Android permissions.

3 RELATED WORK

This section briefly summarizes the state-of-the-art on anti-repackaging based on internal AT checks (Section 3.1), and on anti-virtualization techniques (Section 3.2).

3.1 Anti-Repackaging

The first anti-repackaging technique for mobile apps was proposed in 2015 by Protsenko et al. in [25]. The main idea is to encrypt the *classes.dex* files in the APK and dynamically decrypt them during the app execution. In 2016, Luo et al. in [18] proposed Stochastic Stealthy Network (SSN), which injects a set of detection nodes into the app source code (i.e., through the Java bytecode). The detection nodes hide a stochastic function that aims to detect tampering and, in case, force the app to crash. In 2017, Song et al. [32] proposed an app reinforcing framework named AppIs. The main idea is to create a graph of security units (i.e., detection nodes), which performs i)

AT controls to detect repackaging and ii) integrity checks of other security units. In 2018, Chen et al. [8] proposed a Self-Defending Code (SDC) scheme, encrypting pieces of code. The ciphered portion of code is decrypted and executed at runtime only if the app is not repackaged. Each piece of code is encrypted with a different key, which is related to the AT controls that are activated at runtime. In the same year, Zeng et al. [39] designed BombDroid, a defending technique that leverages logic bomb as an anti-repackaging protection for Android apps. Similar to SDC, BombDroid hides different AT controls inside the encrypted code of a logic bomb. In 2019, Tanner et al. [33] proposed an extension of BombDroid. The main advancement is related to implementing the logic bombs (i.e., AT controls and original code) in the native code. In 2020, Merlo et al. [22] defined some guidelines for reliable anti-repackaging techniques and proposed a new anti-repackaging methodology, i.e., ARMAND [21], that leverages a pseudo-stochastic criterion to inject multiple types of logic bombs both in the Java bytecode and in the native code.

All the proposed techniques allow protecting apps from traditional repackaging to different extents, but are ineffective against repackaging through virtualization-based techniques, as we will discuss in Section 4.

3.2 Anti-Virtualization

Malicious exploitation of the virtualization technique [9, 29, 41, 42] has motivated researchers towards the design of possible defense mechanisms [9, 19, 29, 30, 38, 41], which, to some extent, might also defend from virtualization-based repackaging attacks. Most of them are designed to detect whether an app is running in a virtual environment. To achieve this aim, such solutions are supposed to be included as a library that evaluates specific features of the protected app at runtime. A virtual environment provides the plugin with a different context with respect to the native Android OS that affects the following elements: permissions, number and names of the components, processes names, organization of the internal storage of a plugin, and private data sharing among plugins. For example, a plugin can be granted more permissions than the declared ones, since the sharing of the UID between the container and the plugin also involves the sharing of permissions. Another example refers to the number and names of the components declared by the container, which uses stub components to wrap the plugin ones. This design implies that the Android OS is only aware of the existence of the container components (i.e., the components of the plugins get unnoticed). All existing defense mechanisms against the virtualization technique check the above-mentioned features by either evaluating the return value of specific Android APIs or by inspecting the OS itself (e.g., the `proc` directory), and force the plugin to crash if the return values or the Android OS features are not the expected ones.

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

So far, only two solutions [30, 41] have been specifically designed to detect virtualization-based repackaging attacks. Both of them share the same purpose: first, identifying whether an app uses the virtualization technique and, then, whether its usage is for malicious purposes. Both solutions rely on a static analysis approach, even though the specific detection mechanisms differ from each other. The authors of [41] identify a virtual environment according to the similarity among the declared components (containers usually declare components - named *stub* components - having a very repetitive structure). On the contrary, VAHunt [30] relies on control-flow graphs to see whether a plugin component is eventually replaced by a stub one. Both approaches also differ in the way they recognize a malicious use of virtualization: while the authors of [41] search for a mismatch in the signature of both the plugin and the container, VAHunt checks whether there is any stealthy loading of code.

Even though addressing virtualization-based repackaging attacks, the above-mentioned solutions share common limitations. First, being static analysis tools, they are not able to evaluate code that is dynamically loaded at runtime, although they can detect the usage of the dynamic loading technique. In addition, their usage requires a non-negligible manual effort for the setup and configuration, thus limiting their adoption in the wild. Finally, state-of-the-art approaches focus on detecting already repackaged apps, i.e., the ones equipped with anti-repackaging mechanisms that attackers have successfully bypassed. On the contrary, MARVEL aims to prevent repackaging attempts by protecting the apps.

4 ASSUMPTIONS AND REQUIREMENTS

In this section, we present a threat model (Section 4.1) for both traditional and virtualization-based repackaging attacks. Then, we define a set of security requirements (Section 4.2) that a reliable anti-repackaging scheme should meet.

4.1 Threat model

The goal of a repackaging attack is to distribute a modified version of an app that contains a malicious payload. The victim is a general Android user, daily enjoying her mobile device and regularly downloading apps from any app market (e.g., Google Play Store and Amazon Appstore). The malicious payload can be injected in the victim app through different methodologies according to the approach chosen by the attacker: traditional repackaging or repackaging through the virtualization technique. To achieve her purpose, we assume the attacker can own a mobile device, eventually a rooted/customized one, and can rely on static and dynamic analysis techniques, as well as network traffic analysis ones, to inspect the behavior of the victim app.

The traditional repackaging approach encompasses the steps illustrated in Fig. 2. The attacker first downloads the target APK and analyzes it through static and dynamic techniques (Step 1). Then, she searches for any anti-repackaging protection in the target app (Step 2). If at least a match is found, the attacker needs to try deactivating the protections (Step 3). Otherwise, she can directly move to Step 4 and inject some malicious code into the app. Next, the attacker extensively tests the repackaged app to check whether it works properly (Step 5). In case it works, she can redistribute the repackaged app (Step 6) on the app store. Otherwise, she needs to carry out further analysis, and repeat Steps 1 to 5 (also known as the "try and error cycle"), until she obtains a working repackaged app.

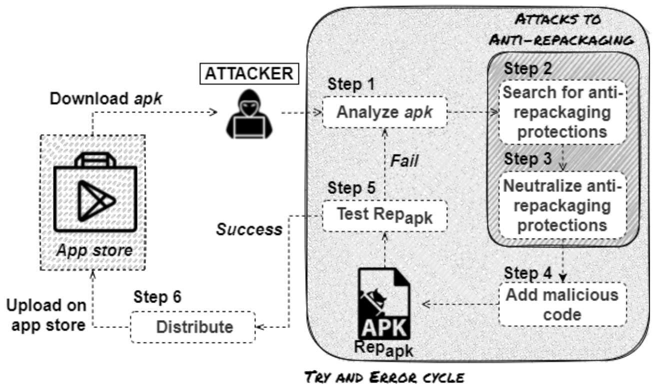


Figure 2: Traditional app repackaging attack.

A virtualization-based repackaging attack is depicted in Fig. 3. This attack is far less complex than the traditional one and much more powerful. A container can execute any external app as a plugin without modifying it, thus passing undetected to traditional AT controls and logic bombs. Furthermore, its proxy capabilities allow intercepting all the API invocations to the Android OS and tamper with the responses, thus overcoming the existing anti-virtualization solutions [1].

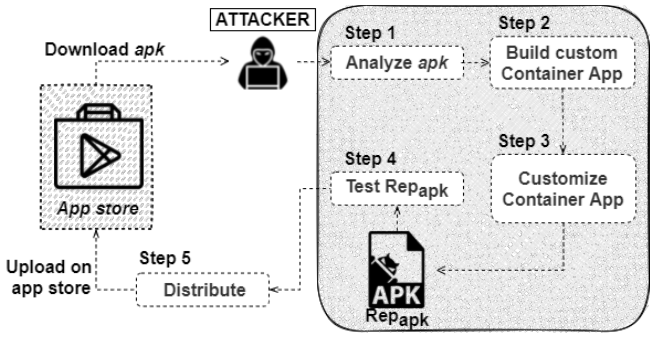


Figure 3: Virtualization-based repackaging attacks.

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

4.2 Security Requirements

The inner design of the virtualization-based attacks allows bypassing all existing defense solutions: similarity checks fail since the malicious container dynamically loads the victim app at runtime, thus bypassing any static analysis. Similarly, anti-repackaging solutions fail since the malicious container i) does not modify the code of the victim app, thus neglecting static AT checks (e.g., signature verification), and ii) can intercept and tamper with any interaction with the Android OS, bypassing dynamic environment controls (e.g., emulator and root detection).

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

- (R1) Preventing the attacker from being able to statically analyze an app to fully reconstruct both the code and the protection controls.
- (R2) Preventing an app from being executed in a malicious container that bypasses traditional AT checks.
- (R3) Detecting an intermediate malicious container that is run in the container's virtual environment and executes a plugin.

5 THE MARVEL PROTECTION SCHEME

MARVEL is an anti-repackaging protection scheme that leverages the virtualization technique to prevent both traditional and virtualization-based repackaging attacks. Furthermore, MARVEL fulfills the security requirements discussed in Section 4.

In this section, we first provide an overview of MARVEL (Section 5.1) and, then, we detail its building blocks, namely code splitting (Section 5.2), Interconnected Anti-Tampering (IAT) controls (Section 5.3) and runtime execution (Section 5.4).

5.1 Overview

MARVEL encompasses a mobile device, running any recent version of the Android OS (i.e., Android API 26+), and a container. Furthermore, MARVEL requires that the protected apps can be successfully executed only as plugins in a trusted virtual container, which is responsible for unlocking the anti-repackaging protections. In our scenario, we suppose that the container is a service that does not require root privileges, like Google Play Services (GPS) [13]: during the first execution, an app verifies if the trusted container is installed (e.g., apps use the `isGooglePlayServicesAvailable`[2] method to verify the availability of GPS) and, if this is the case, it requires to be executed by the trusted container. Mobile vendors already adopted a similar approach. For instance, Samsung Knox [28] is a commercial solution for Mobile Device Management that exploits a pre-installed trusted app that manages corporate data and apps.

MARVEL protects plugins through two approaches: *code splitting* and *IAT*. The former allows to remove portions of code from the original app, thus introducing a mitigation against static analysis inspection (i.e., requirement *R*1). The latter involves the injection of controls which are evaluated during the interaction between the container and the plugin. Thus, if either the plugin is not executed in the expected container (i.e., requirement *R*2), or the container cannot communicate with the intended plugin (i.e., requirement *R*3), the controls fail.

[2]https://developers.google.com/android/reference/com/google/android/gms/common/GoogleApiAvailability

Furthermore, MARVEL enables the use of combined protection patterns, e.g., the injection of an IAT in the code of a method extracted using code splitting. Such a composition increases the difficulty in bypassing the protections, as the attacker is forced to execute and reconstruct the entire package before trying to locate and deactivate all IAT controls.

5.2 Code Splitting

Code splitting techniques allow the partitioning of computations of programs on different nodes to relieve resource-constrained nodes from heavy calculations [26] or to avoid the disclosure of sensitive code/results [2]. In particular, code splitting is well suited for a client-server environment, where the remote server is usually the preferred node to carry out sensitive computations. At the same time, only authenticated clients are allowed to access the results [7, 11].

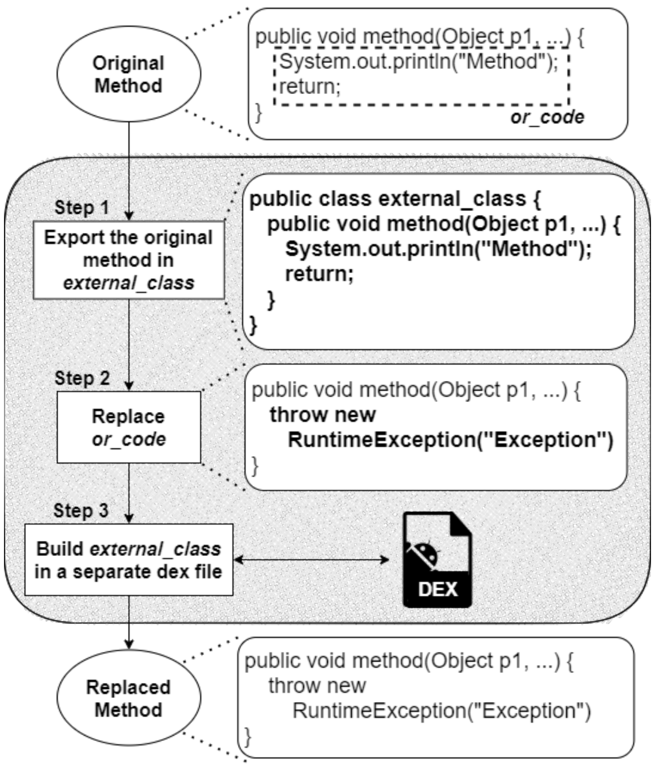
MARVEL applies code splitting between the trusted container and the plugin to fulfill the *R1* requirement. As depicted in Fig. 4, MARVEL selects some methods of the plugin and copies them in a separate external class (e.g., `external_class` in Step 1). Then, MARVEL replaces the body of the selected methods (i.e., the `or_code`) with a `RuntimeException` (Step 2) and it compiles each extracted method in an external dex file (Step 3). At the end of the procedure, there are as many distinct dex files as the number of removed methods. The dex files (i.e., the `external_class`-es) are not saved in the plugin apk, as they are directly dispatched to the trusted container through a secure channel (e.g., downloaded in an encrypted form from a remote, trusted server) during the installation of the plugin. At runtime, the container restores the original code of the removed methods only on-demand (i.e., before their execution). Thus, an attacker should extensively execute the protected app to trigger all removed methods, retrieve their external dex files, and reconstruct the original apk.

5.3 Interconnected Anti-Tampering Controls

To address both the *R2* and the *R3* requirements, MARVEL extends the traditional AT controls to protect both the container and the plugin from repackaging attacks. In particular, we designed IAT, a novel integrity control system based on a challenge-response protocol between the container and the plugin, which leverages the intrinsic role of the container to work as a proxy between the plugin and the Android OS.

MARVEL supports two different types of IAT: *base IAT* and *IAT with encryption.* In the former, the plugin verifies the response from the container against a set of pre-computed hashes; in the latter, the plugin uses the response from the container as a decryption key for a ciphered portion of its code.

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


Original Method
Step 1
Export the original method in *external_class*
Step 2
Replace *or_code*
Step 3
Build *external_class* in a separate dex file
DEX
Replaced Method
or_code

```
public void method(Object p1, ...) {
  System.out.println("Method");
  return;
}
```

```
public class external_class {
  public void method(Object p1, ...) {
    System.out.println("Method");
    return;
  }
}
```

```
public void method(Object p1, ...) {
  throw new
    RuntimeException("Exception")
}
```

```
public void method(Object p1, ...) {
  throw new
    RuntimeException("Exception")
}
```

Figure 4: Code splitting transformation process.

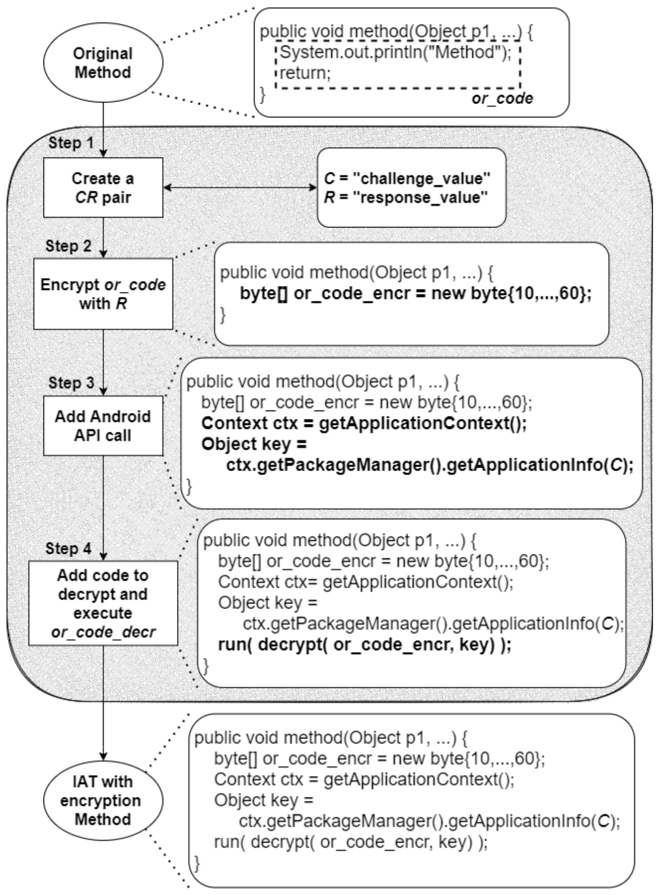
Fig. 5 shows the transformation steps involved in the creation of an IAT with encryption. In particular, MARVEL first generates a challenge-response pair (Step 1). Then, it replaces the original code of the method with its encrypted form generated using `R` as the encryption key (Step 2). The procedure injects the challenge vector (i.e., an Android API such as `getPackageInfo`) in the body of the method (Step 3), which contains the challenge value (`C`) as a parameter. Finally, MARVEL adds the logic to perform the decryption with the response value sent by the container (i.e., `key`), and the execution of the decrypted code (Step 4).

Fig. 6 depicts the injection process for a base IAT. Differently from the previous case, MARVEL does not encrypt the `or_code`. Furthermore, the procedure adds the logic to verify the response obtained by the container against a pre-generated hash of *R* (i.e., $HASH_R$) (Step 3). Similar to logic bombs, the protection of a base IAT is granted by the one-way property of cryptographic hash functions, which makes it extremely hard for the attacker to statically retrieve the original value (i.e., the response) from its hash.

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

5.4 Runtime Execution

Fig. 7 summarizes the MARVEL architecture and the interactions between the trusted container and the protected plugin. At the startup of the plugin, the container verifies some plugin metadata



```
public void method(Object p1, ...) {
  System.out.println("Method");
  return;
}
```

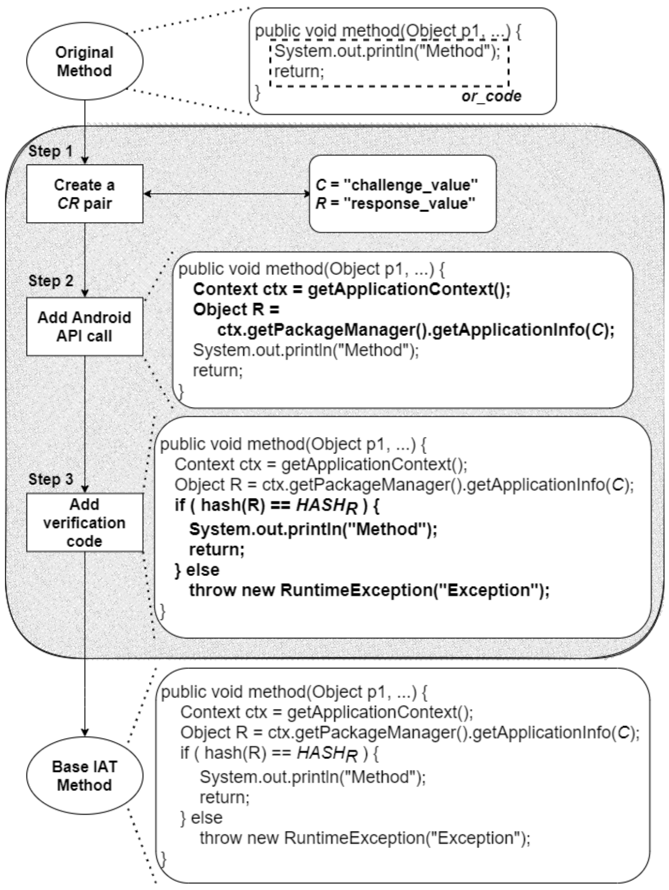
```
public void method(Object p1, ...) {
    byte[] or_code_encr = new byte{10,...,60};
}
```

```
public void method(Object p1, ...) {
  byte[] or_code_encr = new byte{10,...,60};
  Context ctx = getApplicationContext();
  Object key =
      ctx.getPackageManager().getApplicationInfo(C);
}
```

```
public void method(Object p1, ...) {
  byte[] or_code_encr = new byte{10,...,60};
  Context ctx= getApplicationContext();
  Object key =
      ctx.getPackageManager().getApplicationInfo(C);
  run( decrypt( or_code_encr, key) );
}
```

```
public void method(Object p1, ...) {
  byte[] or_code_encr = new byte{10,...,60};
  Context ctx = getApplicationContext();
  Object key =
      ctx.getPackageManager().getApplicationInfo(C);
  run( decrypt( or_code_encr, key) );
}
```

Figure 5: IAT with encryption transformation process.



```
public void method(Object p1, ...) {
  System.out.println("Method");
  return;
}
```

```
public void method(Object p1, ...) {
  Context ctx = getApplicationContext();
  Object R =
      ctx.getPackageManager().getApplicationInfo(C);
  System.out.println("Method");
  return;
}
```

```
public void method(Object p1, ...) {
  Context ctx = getApplicationContext();
  Object R = ctx.getPackageManager().getApplicationInfo(C);
  if ( hash(R) == HASH_R ) {
    System.out.println("Method");
    return;
  } else
    throw new RuntimeException("Exception");
}
```

```
public void method(Object p1, ...) {
  Context ctx = getApplicationContext();
  Object R = ctx.getPackageManager().getApplicationInfo(C);
  if ( hash(R) == HASH_R ) {
    System.out.println("Method");
    return;
  } else
    throw new RuntimeException("Exception");
}
```

Figure 6: Base IAT transformation process.

(e.g., the signature of the APK file). If the checks fail, the container does not execute the plugin. In the same way, the plugin carries out, during its execution, an evaluation of the running virtual environment, e.g., by checking the class name of the proxy object injected by the container. If this check fails, the app crashes immediately. Such integrity controls enable a further detection of a tampered environment, preventing the runtime execution of the plugin in an untrusted - or malicious - container. Afterwards, the MARVEL container manages the additional anti-repackaging controls, i.e. code splitting and IAT.

Code Splitting. At runtime, the container detects whenever the plugin loads a fake method and replaces it with the correct one. To this aim, the container injects a Custom ClassLoader (CCL) in the plugin process to intercept the loading of any new class in the current JVM memory. Once the loading event occurs (i.e., method M_i), the CCL resolves the requested class and, if it contains a fake method, the CCL injects the original one (i.e., retrieved from the `dex` file DEX_i) in the plugin memory. This operation is transparent to the plugin, which receives the resolved method (i.e., CM_i).

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

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

6 IMPLEMENTATION OF MARVEL

We implemented the MARVEL protection scheme in a prototype tool, called MARVELoid, which injects the protection in the plugins (i.e., Section6.1). Furthermore, we extended the VirtualApp framework [17] to develop the trusted container responsible for the enforcement of the runtime protection (i.e., Section 6.2). The source code of MARVELoid and the trusted container are available at https://github.com/totoR13/MARVEL. It is worth highlighting that,

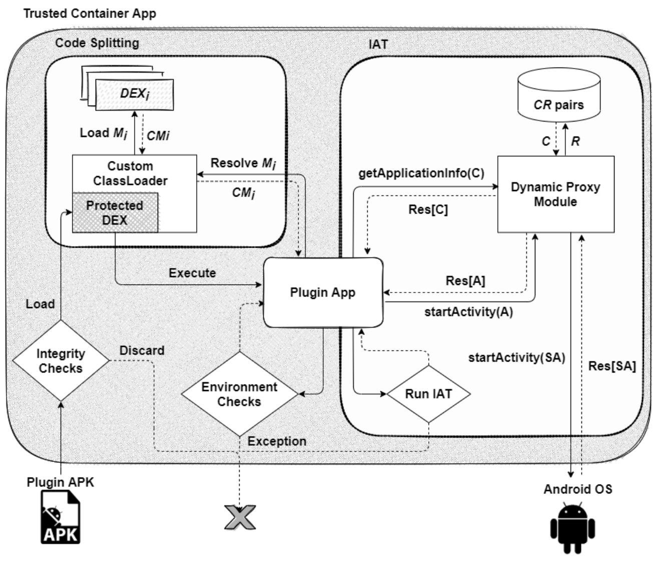


Figure 7: The MARVEL architecture.

given the common design, MARVEL can be ported on DroidPlugin, as well.

6.1 MARVELoid

MARVELoid is a Java-based tool devoted to the generation of the protected plugin starting from the original app. The tool leverages the Soot framework [14] to analyze and modify the bytecode of the Android app, and Jarsigner [24] to sign the output APK file with a certificate.

MARVELoid requires three input parameters, namely the probability to i) replace a method through code splitting (P_{repl}), ii) inject a base IAT control (P_{base_IAT}), and iii) inject an encrpyted IAT (P_{enc_IAT}). Overall, the probability to introduce a protection is given by:

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

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

Also, MARVELoid requires the Package Name (PN) that identifies the part of the app (i.e., the group of classes) to include in the protection process. The default value of PN is the app package name, which is specified in the `AndroidManifest.xml` file of the APK[3].

Given an app, MARVELoid unpacks the APK file to extract the `classes.dex` file(s) and translates the Java bytecode into an intermediate representation (i.e., Jimple) by leveraging the Soot framework. Then, the tool scans the code and, for each method, computes the probability to protect it according to the input percentages. For each method, MARVELoid calculates the chance to inject each protection (i.e., code splitting, base IAT, and IAT with encryption). To do so, it creates three objects that extends the `BodyTransformer`[4] class. These objects execute concurrently and compute the probability to protect a method with their specific protection type. In this way, according to the input probability, a method can be protected with multiple protection types in a random order, depending on the scheduling of the objects.

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

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

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

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

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

6.2 Trusted Container

The trusted container is an extension of the VirtualApp framework in charge of offering the virtualized environment to execute the plugins and enforcing the MARVEL protections.

In detail, the container i) performs the integrity checks of the plugin, ii) interacts with the plugin through the IAT, and iii) restores the original content of the fake methods.

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

Code Splitting. To keep track of the loading of a fake method, the container injects a custom ClassLoader (CCL) into the process

[3]https://developer.android.com/guide/topics/manifest/manifest-intro
[4]https://javadoc.io/doc/ca.mcgill.sable/soot/latest/soot/BodyTransformer.html

of the plugin. The CCL is an instance of the `dalvik.system.DexClassLoader` and overloads the `loadClass` method. In particular, the CCL i) resolves the requested class, ii) checks if it contains some fake methods, and iii) in case of fake methods, it replaces them with the correct ones.

The container leverages the DCL (i.e., Dynamic Code Loading) to load the original instructions into the plugin process and ART Instrumentation [4] to inject the correct bytecode. To implement the ART Instrumentation inside the trusted container, we leveraged the YAHFA hooking framework[5].

With the support of YAHFA, we can overcome the limitations of the Java Dynamic Proxy that restricts the hooking only to object instances of classes with at least one interface. In detail, ART Instrumentation allows the modification of the ArtMethod[6] entry points. The container overrides the value of the `entryPointFromQuickCompiledCode`, which is the pointer to the compiled code of the interpreter (i.e., the `dex` file of the plugin), with a pointer to the loaded code of the replaced method (included in a `dex` file during the MARVELoid process).

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

7 EXPERIMENTAL EVALUATION

We empirically assessed the performance of the MARVEL methodology by i) applying MARVELoid over a dataset of 4000 apps (downloaded from the Google Play Store in December 2020) to evaluate the distribution of the protection values according to several combinations of input percentages, and ii) executing a subset of 45 randomly-selected apps (with the best sets of input parameters) to evaluate their compatibility with virtualized environments and the introduced runtime overhead.

The apps of the dataset have an average rating of 4 stars on the Google Play Store and belong to more than 30 different categories. Furthermore, 60% of them have the `minSDK` above Android 4.0 to include the most recent API features.

Static evaluation. The experiments were hosted on a virtual machine running Ubuntu 20.04 with 8 processors and 32GB RAM. All experiments were conducted using the default value of the `PN` input parameter of MARVELoid (i.e., the package name attribute of the `AndroidManifest.xml` file). We conducted tuning tests to detect the on-average best combinations of input percentages (i.e., P_{repl}, P_{enc_AT}, and P_{base_IAT}) to ensure a reasonable trade-off between the protection overhead values (i.e., space and time overhead) and the protection level (i.e., the number of injected protections). In particular, we computed the protection overhead and protection values for 24 combinations (i.e., the permutations of [5, 10, 20, 30]).

MARVELoid was able to apply 96000 protections (i.e., 4k apps for each permutation) in nearly 109 days and 14 hours (i.e., 9477578 seconds) of computation. The protection of a single app took, on average, 98 seconds.

MARVELoid worked successfully in the 97.3% of the cases (i.e., it generated a valid protected APK). The remaining 2.7% (i.e., 2624) failed due to well-known bugs of the adopted libraries (such as a Soot bug[7]) or crashes during the tool transformation process.

Fig. 8 shows the distribution of the success percentage for each permutation. Although the value slightly decreases with higher values of P_{repl}, it is worth noticing that its range always sits between 96% and 98%.

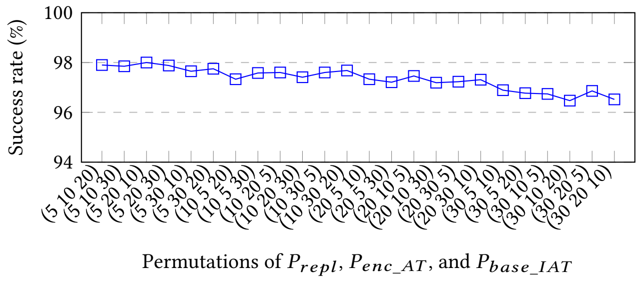


Figure 8: Percentage of apps successfully protected by MARVELoid.

Then, we calculated the average percentage of space overhead of the protected APK compared with the original one, and for each permutation. Fig. 9 shows the results for the 24 combinations of the input values. The value distribution suggests that the space overhead is directly proportional to P_{enc_IAT}. Such behavior is reasonable because MARVELoid adds some auxiliary code to handle the decryption and execution of the encrypted IAT. All in all, the space overhead introduced by the protection is always less than 18% in the worst case.

Finally, Fig. 10 shows the average number and distribution of the injected protections (i.e., the average number of replaced methods, IAT with encryption and base IAT) for each permutation of the input probability values. During the experimental activities, MARVELoid was able to inject a minimum of 68 and a maximum of 135 protection elements in the protected apps.

Runtime evaluation. The last set of experiments is aimed at evaluating the effectiveness and usability of the proposed protection scheme. To do so, we randomly selected 45 apps from the dataset of 4k apps that reported at least 5.000.000 downloads on the Google Play Store in December 2020.

In the first part of the runtime evaluation, we assessed the compatibility of traditional Android apps w.r.t. virtualization, as well as the overhead introduced in terms of CPU and memory usage.

[5]https://github.com/PAGalaxyLab/YAHFA

[6]https://chromium.googlesource.com/android_tools/+/78ccfd5f7e4880597fd90c61453a3be0e7aee5f0/sdk/sources/android-21/java/lang/reflect/ArtMethod.java

[7]https://github.com/soot-oss/soot/issues/1474

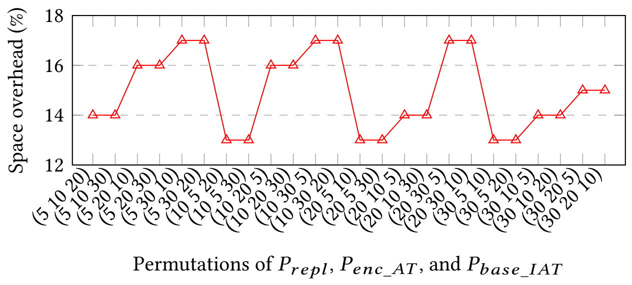


Figure 9: Space overhead introduced by MARVELoid.

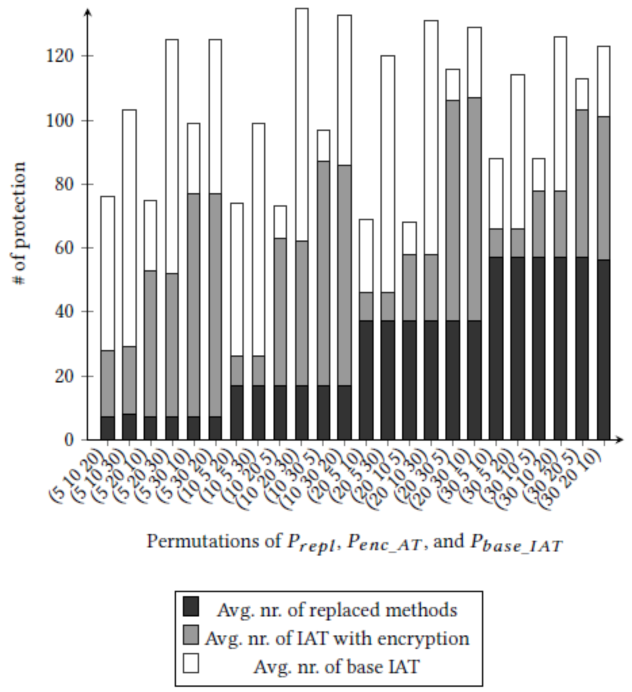


Figure 10: Distribution of the average protection values inserted by MARVELoid.

Then, we evaluated i) the effectiveness of MARVELoid by running the protected apps inside the Trusted Container, and ii) the corresponding overhead (using the results of the standard virtualization as reference values).

For the testing phase, we used a set of emulated Android 8.0 devices equipped with a dual-core processor and 2GB of RAM and ARES [27], a black-box tool that leverages Deep Reinforcement Learning to stimulate Android apps automatically.

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

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

The protection phase introduced, on average, 111 protections (i.e., 32 replaced methods, 38 IAT with encryption, and 41 base IAT) for $Setup_{10\%}$ and 175 protections (i.e., 51 replaced methods, 58 IAT with encryption, and 66 base IAT) for the second one.

For each combination, we executed each app 10 times for 4 minutes to verify its correct execution inside the Trusted Container and the introduced overhead.

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

- 8 apps ($Setup_{10\%}$) and 7 apps ($Setup_{15\%}$) crashed due to a well-known Soot bug[8]. The bug affects the transformation process from the bytecode to Jimple of third-party libraries included in APK files, causing the final app to crash during the execution.
- 4 apps ($Setup_{10\%}$) and 4 apps ($Setup_{15\%}$) triggered a timeout defined by the UiAutomator2 library, used by ARES to test the application. Unfortunately, after a deeper investigation, we discovered that such timeout is hardcoded and cannot be modified[9]. It is worth noticing that these apps did not crash due to the protection but were terminated by the library.
- Finally, 6 apps ($Setup_{10\%}$) and 4 apps ($Setup_{15\%}$) threw new exceptions directly related to the protection process.

Concerning the last point, the root cause analysis of the crashes caused by MARVELoid led to the discovery of two issues of our prototype implementation. The first one is related to an incorrect referencing of the Context[10] of the app. In several cases, the plugin raised a null pointer exception due to the absence of the correct reference. The second bug occurs if a replaced method is already loaded in the CCL. In such a case, a crash can occur because the CCL does not inject the correct body of the method, causing a runtime exception.

The dynamic testing phase allowed us to monitor the overhead introduced by both traditional virtualization and our solution on the set of 45 apps. Table 1 shows the minimum, average, and maximum overhead values expressed in percentage points (pp). In the first column, we compared the execution of apps in the virtual environment with the traditional execution method. For the protected apps (i.e., second and third columns), we computed the overhead compared to the execution in a clean virtual environment. In this

[8]See the related Github issues https://github.com/soot-oss/soot/issues/1151, and https://github.com/soot-oss/soot/issues/1615
[9]https://github.com/appium/appium/issues/12555
[10]https://developer.android.com/reference/android/content/Context

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

Table 1: CPU and memory usage overhead in percentage point (pp).

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

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

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

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

8 DISCUSSION AND CONCLUSION

In this paper, we proposed MARVEL, a methodology that allows the protection of Android apps against both traditional and virtualization-based repackaging attacks. We implemented MARVELoid as a solution to inject MARVEL's protections inside apps, and we customized the VirtualApp framework to provide a Trusted Container for the enforcement of the runtime protection.

From a security point of view, MARVEL is effective. With respect to the repackaging attacks presented in Section 4.1, MARVEL provides a preventive anti-repackaging protection in both repackaging attack scenarios (i.e., traditional and through virtualization).

First of all, the analysis of the victim app APK (i.e., Step 1 in Fig. 2) becomes more challenging for the attacker due to the code splitting and the IAT with encryption, which remove or encrypt some portions of code from the protected app. Concerning the identification and neutralization of the repackaging protections, the victim app might be equipped with (Step 2 and Step 3 in Fig. 2), the mutual collaboration between the Trusted Container and the plugin makes this goal harder to achieve for the attacker: she has to investigate the runtime communication between the Trusted Container and plugin through dynamic analysis techniques, to retrieve the *CR* pairs or the original code after code splitting. Moreover, since a method can be concurrently transformed by several protection mechanisms (e.g., an extracted method can contain an IAT), the attacker has to recursively resolve the nested protection, once she has disabled the external one. Finally, the customization of a container to setup a virtualization-based repackaging attack (Step 2 and Step 3 in Fig. 3) is prevented by MARVEL thanks to the mutual integrity checks between the Trusted Container and the plugin, which stops the execution of the latter in case the environment is detected to be not secure. Overall, MARVEL makes virtualization-based anti-repackaging attacks as much complex as traditional ones.

We designed MARVELoid to minimize the impact on the end-user by only requiring the device to be equipped with a TC, without any modification to the underlying OS. The user could either find the TC pre-installed on the device, similarly to other virtualization technologies, or download it from a store. In the same way, the user could have the plugin app embedded in the TC or download it from an app store. From an experimental point of view, our evaluation of MARVEL over 4.000 Android apps demonstrated the applicability and efficacy of the tool and the proposed protection scheme.

As a future extension of this work, we plan to i) extend the communication mechanism between the plugin and container, ii) include additional AT controls inside the IAT, and iii) enhance the code splitting technique to decrease the CPU overhead.

REFERENCES

[1] Marco Alecci, Riccardo Cestaro, Mauro Conti, Ketan Kanishka, and Eleonora Losiouk. 2020. Mascara: A Novel Attack Leveraging Android Virtualization. arXiv:2010.10639 [cs.CR]

[2] Viticchié Alessio, Regano Leonardo, Basile Cataldo, Torchiano Marco, Ceccato Mariano, and Tonella Paolo. 2020. Empirical assessment of the effort needed to attack programs protected with client/server code splitting. *Empirical Software Engineering* 25, 1 (2020), 1–48.

[3] Avast. 2016. Malware posing as dual instance app steals users' Twitter credentials. https://blog.avast.com/malware-posing-as-dual-instance-app-steals-users-twitter-credentials Accessed online: December 3, 2021.

[4] Michael Backes, Sven Bugiel, Oliver Schranz, Philipp Von Styp-Rekowsky, and Sebastian Weisgerber. 2017. ARTist: The Android Runtime Instrumentation and Security Toolkit. In *2017 IEEE European Symposium on Security and Privacy (EuroS P)*. 481–495. https://doi.org/10.1109/EuroSP.2017.43

[5] BBC. 2017. Fake WhatsApp app downloaded more than one million times. https://www.bbc.com/news/technology-41886157 Accessed online: December 3, 2021.

[6] Stefano Berlato and Mariano Ceccato. 2020. A large-scale study on the adoption of anti-debugging and anti-tampering protections in android apps. *Journal of Information Security and Applications* 52 (2020), 102463. https://doi.org/10.1016/j.jisa.2020.102463

[7] Mariano Ceccato, Mila Dalla Preda, Jasvir Nagra, Christian Collberg, and Paolo Tonella. 2007. Barrier Slicing for Remote Software Trusting. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*. 27–36. https://doi.org/10.1109/SCAM.2007.27

[8] K. Chen, Y. Zhang, and P. Liu. 2018. Leveraging Information Asymmetry to Transform Android Apps into Self-Defending Code Against Repackaging Attacks. *IEEE Transactions on Mobile Computing* 17, 8 (2018), 1879–1893.

[9] Deshun Dai, Ruixuan Li, Junwei Tang, Ali Davanian, and Heng Yin. 2020. Parallel Space Traveling: A Security Analysis of App-Level Virtualization in Android. In *Proceedings of the 25th ACM Symposium on Access Control Models and Technologies (SACMAT '20)*. Association for Computing Machinery, New York, NY, USA, 25–32. https://doi.org/10.1145/3381991.3395608

[10] Google Developers. 2020. Enable multidex for apps with over 64K methods. https://developer.android.com/studio/build/multidex Accessed online: December 3, 2021.

[11] Lynn Futcher and Rossouw von Solms. 2008. Guidelines for Secure Software Development. In *Proceedings of the 2008 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*

on IT Research in Developing Countries: Riding the Wave of Technology (SAICSIT '08). Association for Computing Machinery, New York, NY, USA, 56–65. https://doi.org/10.1145/1456659.1456667

[12] Hugo Gonzalez, Natalia Stakhanova, and Ali A. Ghorbani. 2015. DroidKin: Lightweight Detection of Android Apps Similarity. In *International Conference on Security and Privacy in Communication Networks*, Jing Tian, Jiwu Jing, and Mudhakar Srivatsa (Eds.). Springer International Publishing, Cham, 436–453.

[13] Google. 2021. Overview of Google Play services. https://developers.google.com/android/guides/overview. Accessed online: December 3, 2021.

[14] Sable Research Group. 2021. Soot - A Java optimization framework. https://github.com/soot-oss/soot Accessed online: December 3, 2021.

[15] ImaTech Innovations. 2020. Parallel Accounts. https://play.google.com/store/apps/details?id=com.in.parallel.accounts Accessed online: December 3, 2021.

[16] Li Li, Tegawendé F. Bissyandé, and Jacques Klein. 2021. Rebooting Research on Detecting Repackaged Android Apps: Literature Review and Benchmark. *IEEE Transactions on Software Engineering* 47, 4 (2021), 676–693. https://doi.org/10.1109/TSE.2019.2901679

[17] Jining Luohe Network Technology Co. Ltd. 2020. VirtualApp. https://github.com/asLody/VirtualApp Accessed online: December 3, 2021.

[18] L. Luo, Y. Fu, D. Wu, S. Zhu, and P. Liu. 2016. Repackage-Proofing Android Apps. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 550–561.

[19] Tongbo Luo, Cong Zheng, Zhi Xu, and Xin Ouyang. 2017. Anti-Plugin: Don't let your app play as an Android plugin. In *Proceedings of Blackhat Asia 2017*.

[20] Zhuo Ma, Haoran Ge, Yang Liu, Meng Zhao, and Jianfeng Ma. 2019. A Combination Method for Android Malware Detection Based on Control Flow Graphs and Machine Learning Algorithms. *IEEE Access* 7 (2019), 21235–21245. https://doi.org/10.1109/ACCESS.2019.2896003

[21] Alessio Merlo, Antonio Ruggia, Luigi Sciolla, and Luca Verderame. 2021. ARMAND: Anti-Repackaging through Multi-pattern Anti-tampering based on Native Detection. *Pervasive and Mobile Computing* 76 (2021), 101443. https://doi.org/10.1016/j.pmcj.2021.101443

[22] Alessio Merlo, Antonio Ruggia, Luigi Sciolla, and Luca Verderame. 2021. You Shall not Repackage! Demystifying Anti-Repackaging on Android. *Computers & Security* 103 (2021), 102181. https://doi.org/10.1016/j.cose.2021.102181

[23] Oracle. 2021. Dynamic Proxy Classes. https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html Accessed online: December 3, 2021.

[24] Oracle. 2021. Jarsigner. https://docs.oracle.com/javase/7/docs/technotes/tools/windows/jarsigner.html Accessed online: December 3, 2021.

[25] Mykola Protsenko, Sebastien Kreuter, and Tilo Müller. 2015. Dynamic Self-Protection and Tamperproofing for Android Apps Using Native Code. In *2015 10th International Conference on Availability, Reliability and Security*. 129–138. https://doi.org/10.1109/ARES.2015.98

[26] Bharat S. Rawal, Ramesh K. Karne, and Alexander L. Wijesinha. 2012. Split protocol client/server architecture. In *2012 IEEE Symposium on Computers and Communications (ISCC)*. 000348–000353. https://doi.org/10.1109/ISCC.2012.6249320

[27] Andrea Romdhana, Alessio Merlo, Mariano Ceccato, and Paolo Tonella. 2022. Deep Reinforcement Learning for Black-Box Testing of Android Apps. *ACM Transactions on Software Engineering and Methodology* (2022).

[28] Samsung. 2021. Samsung Knox. https://www.samsungknox.com/en Accessed online: December 3, 2021.

[29] Luman Shi, Jianming Fu, Zhengwei Guo, and Jiang Ming. 2019. "Jekyll and Hyde" is Risky: Shared-Everything Threat Mitigation in Dual-Instance Apps. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '19)*. Association for Computing Machinery, New York, NY, USA, 222–235. https://doi.org/10.1145/3307334.3326072

[30] Luman Shi, Jiang Ming, Jianming Fu, Guojun Peng, Dongpeng Xu, Kun Gao, and Xuanchen Pan. 2020. VAHunt: Warding Off New Repackaged Android Malware in App-Virtualization's Clothing. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20)*. Association for Computing Machinery, New York, NY, USA, 535–549. https://doi.org/10.1145/3372297.3423341

[31] Vikas Sihag, Manu Vardhan, and Pradeep Singh. 2021. A survey of android application and malware hardening. *Computer Science Review* 39 (2021), 100365. https://doi.org/10.1016/j.cosrev.2021.100365

[32] Lina Song, Zhanyong Tang, Zhen Li, Xiaoqing Gong, Xiaojiang Chen, Dingyi Fang, and Zheng Wang. 2017. Appis: Protect android apps against runtime repackaging attacks. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 25–32.

[33] Simon Tanner, Ilian Vogels, and Roger Wattenhofer. 2020. Protecting Android Apps from Repackaging Using Native Code. In *Foundations and Practice of Security*, Abdelmalek Benzekri, Michel Barbeau, Guang Gong, Romain Laborde, and Joaquin Garcia-Alfaro (Eds.). Springer International Publishing, Cham, 189–204.

[34] DroidPlugin Team. 2020. DroidPlugin. https://github.com/DroidPluginTeam/DroidPlugin Accessed online: December 3, 2021.

[35] MA Team. 2021. Multiple Accounts:Parallel App. https://play.google.com/store/apps/details?id=com.excelliance.multiaccounts Accessed online: December 3, 2021.

[36] LBE Tech. 2021. Parallel Space - FMulti Account. https://play.google.com/store/apps/details?id=com.lbe.parallel.intl Accessed online: December 3, 2021.

[37] Ke Tian, Danfeng Yao, Barbara G. Ryder, and Gang Tan. 2016. Analysis of Code Heterogeneity for High-Precision Classification of Repackaged Malware. In *2016 IEEE Security and Privacy Workshops (SPW)*. 262–271. https://doi.org/10.1109/SPW.2016.33

[38] Yifang Wu, Jianjun Huang, Bin Liang, and Wenchang Shi. 2020. Do not jail my app: Detecting the Android plugin environments by time lag contradiction. *Journal of Computer Security* 28 (01 2020), 1–25. https://doi.org/10.3233/JCS-191325

[39] Qiang Zeng, Lannan Luo, Zhiyun Qian, Xiaojiang Du, and Zhoujun Li. 2018. Resilient Decentralized Android Application Repackaging Detection Using Logic Bombs. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO 2018)*. Association for Computing Machinery, New York, NY, USA, 50–61. https://doi.org/10.1145/3168820

[40] Qiang Zeng, Lannan Luo, Zhiyun Qian, Xiaojiang Du, and Zhoujun Li. 2018. Resilient decentralized android application repackaging detection using logic bombs. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. 50–61.

[41] Lei Zhang, Zhemin Yang, Yuyu He, Mingqi Li, Sen Yang, Min Yang, Yuan Zhang, and Zhiyun Qian. 2019. App in the Middle: Demystify Application Virtualization in Android and Its Security Threats. *Proc. ACM Meas. Anal. Comput. Syst.* 3, 1, Article 17 (March 2019). https://doi.org/10.1145/3322205.3311088

[42] Cong Zheng, Tongbo Luo, Zhi Xu, Wenjun Hu, and Xin Ouyang. 2018. Android Plugin Becomes a Catastrophe to Android Ecosystem. In *Proceedings of the First Workshop on Radical and Experiential Security (RESEC '18)*. ACM, New York, NY, USA, 61–64. https://doi.org/10.1145/3203422.3203425