Under the Hood of MARVEL

Antonio Ruggia[a*], Eleonora Losiouk[b], Luca Verderame[a], Mauro Conti[b], Alessio Merlo[a]
[a] DIBRIS — University of Genoa, Genoa, Italy
(antonio.ruggia|luca.verderame|alessio)@dibris.unige.com
[b] University of Padua, Padua, Italy
(eleonora.losiouk|mauro.conti)@unipd.it

***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. To mitigate this issue, we proposed MARVEL [26], the first methodology that allows preventing both traditional and virtualization-based repackaging attacks. MARVEL 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 our protection scheme, we implemented it in two tools: MARVELoid to protect the plugins, and a container app (Trusted Container) that enforces the protection at runtime.**

In this paper, we describe the automatic testing pipeline used in the experimental campaign conducted to evaluate the MARVEL implementation in terms of *correctness*, *performance*, and *effectiveness*. Moreover, we perform an in-depth analysis of the dataset used during the evaluation and discuss the open challenges. The first set of experiments aims to evaluate the protection performed by MARVELoid, verifying its viability and reliability. To do so, we protected 4000 apps with 24 different configurations of the MARVELoid 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. Then, we evaluate the runtime overhead introduced by our solution, which turned out to have a negligible impact in terms of average CPU (<5%) and memory overhead (<0.5%).

I. Introduction

The purpose of Android repackaging attacks is to lure the user into installing a non-legitimate app that looks like the original one. The attacker generates such a malicious app by reverse engineering the target app, modifying its content (e.g., injecting malicious code), recompiling and finally distributing the modified app on the app markets or through other channels. Given the similarity between repackaged and original apps, the user is unable to distinguish between them and can be fooled into installing the malicious version. The repackaged app executes the attacker's code on the victim's phone and it might aim to damage not only the end-user (e.g., stealing sensitive data from the phone, injecting fake one), but also the app developer (e.g., re-distributing paid apps for free, removing/redirecting ads earnings, making the app willingly unusable).

A recent growing trend in repackaging attacks concerns the exploitation of the Android virtualization technique [10] to generate and distribute malicious repackaged apps in an easier way with respect to the traditional 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: it can execute any app in its virtual environment (i.e., even apps that are not installed on the mobile device), 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 — their API calls, passing unnoticed to the apps themselves. To do so, the container leverages its proxy role to perform such modification without modifying the content of the plugin's APK.

The popularity of the virtualization technique, confirmed by the number of downloads of virtualization apps on the Google Play Store [33], [14], [32], is given by its main use case scenario: running multiple instances of the same app on a single device. In this scenario, if a user has two separate Telegram accounts and wants to use them simultaneously, he 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 [38], [37], [28], [10].

With respect to traditional repackaging attacks, virtualization-based ones are easier to setup. The former requires the attacker to i) decompile the APK of the victim app, ii) detect and, in case, remove anti-repackaging protections, iii) inject some malicious code, and iv) recompile it again. On the other hand, 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. Besides creating a virtual environment, a container app can also hijack user input and steal users' sensitive data. The malicious code can be part of the container logic or another plugin running in the same virtual environment as the victim. Two examples of virtualization-based malware have already been detected in the wild [6], [5], which target respectively Twitter and WhatsApp.

To defend against the malicious usage of the virtualization

Learning from Authoritative Security Experiment Results (LASER) 2021
7 December 2021, Virtual
ISBN 1-891562-82-7
https://dx.doi.org/10.14722/laser-acsac.2021.23258
www.acsac.org

technique, researchers have put forward some solutions [34], [28], [10], [37], [18], [29]. 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. The limit of such approaches is twofold: the checks they rely on can be easily bypassed [4], and they are not able to distinguish between benign and malicious usage of the virtualization technique. On the contrary, two solutions [37], [29] have been proposed to defend against virtualization-based repackaging attacks. Both of them rely on a static analysis approach to detect any usage of the virtualization technique and 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 our previous work [26], we proposed 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. Furthermore, we developed MARVEL protection scheme in two prototypes: MARVELoid, a standalone tool to inject anti-repackaging protection, and the Trusted Container (TC), a virtualization-based app that enforces the MARVEL protection at runtime. To evaluate the feasibility of MARVEL, in our previous work [26], we conducted an experimental campaign based on an automatic testing pipeline. In particular, we protected 4000 apps with 24 different configurations of the protection parameters (i.e., *workloads*), 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. Then, 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.

In this work, we present the details of the automatic testing pipeline we designed to conduct the experimental campaign in our previous work [26], as well as the dataset of apps we used for the runtime evaluation. Our testing pipeline aims to evaluate the *correctness*, *performance*, and *effectiveness* of the MARVEL implementation. In particular, we focused on the *repeatability* of the results, i.e. on the process of obtaining the same results as the reported ones through an independent experiment. To this aim, we released both the source code of the developed tools (i.e., MARVELoid, Trusted Container, and the automatic pipeline) and the results of our experimental campaign [1], for which we also obtained the "2021 ACSAC Distinguished Artifact Award" [3]. We believe that releasing software is an important step toward the reproducibility of research results, and it strongly contributes to the deployment of novel security solutions in a real-world scenario.

[1] https://github.com/totoR13/MARVEL

Organization. The rest of the paper is organized as follows: in Section II, we illustrate the threat model considered for the design of MARVEL, together with a motivating example and an overview of MARVEL; Section III presents the goals we wanted to achieve through our testing pipeline as well as the details of the dataset of apps we used; in Section IV we describe the design of the testing pipeline, while in Section V we present the results obtained through the experimental campaign. Finally, we conclude the paper by comparing our testing methodology with the ones presented in previous works in Section VI and with closing remarks and possible future extensions in Section VII.

II. Background

In this section, we briefly summarize the threat model of the repackaging attack, present a motivating example, and introduce the key aspects of the MARVEL protection scheme.

A. Threat Model

The goal of a repackaging attack is to distribute a modified version of an Android 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 App Store). The malicious payload can be injected into 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. On the other hand, to protect from such threats, the apps developers inject detection nodes that implement anti-tampering (AT) controls aimed at checking some metadata of an app (e.g., the signature or the package name).

In traditional repackaging [21], the attacker first downloads the target APK and analyzes it through both static and dynamic techniques. Then, she searches for any anti-repackaging and anti-tampering (AT) protections in the target app. AT controls check some metadata of an app (e.g., the signature or the package name) to detect modifications. If at least a match is found, the attacker needs to try deactivating the protections; otherwise, she can directly inject some malicious code into the app or modify the original one. Next, the attacker extensively tests the repackaged app to check whether it works properly. If this is the case, she can redistribute the repackaged app to an app store. Otherwise, she needs to carry out further analysis and repeat the whole process (also known as the "try and error cycle") until she obtains a working repackaged app.

A virtualization-based repackaging attack is depicted in Fig. 1. 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 tampering with the responses, thus overcoming the existing anti-virtualization solutions [4].

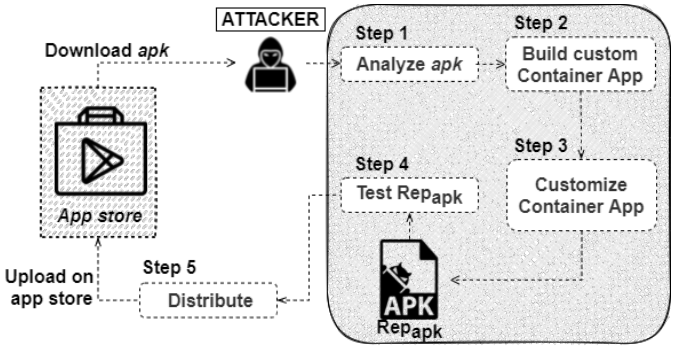


Fig. 1: Virtualization-based repackaging attacks.

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).

B. Motivating Example

To demonstrate the applicability and impact of virtualization-based repackaging attacks, we developed a (malicious) container app of Telegram, one of the most popular messaging app for Android, that is able to retrieve the plain text of received and sent messages. We manually reverse engineer the target apps to identify the Java methods in charge to handle the incoming/outgoing messages and implement a custom hooking function using ART instrumentation [23]. We successfully performed the virtualization-based repackaging attack, retrieving the plain text of all messages. For instance, in Telegram, the `processSendingText` method of `org.telegram.ui.Components.ChatActivityEnterView` class is responsible for retrieving and encrypting the text written by the user, and sending the cipher text over the network. The malicious container hooks such a method and logs the input parameter (i.e., plain text of the outcoming message).

It is worth noticing that a malicious user can create a generic container hooking only Android view items, such as the `TextView`. However, this approach has a limitation since it retrieves all the possible strings shown.

Finally, we point out that, thanks to the Android virtualization, we can overcome the integrity checks performed in the plugin apps. In particular, current 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). For instance, while WhatsApp implements several controls (e.g., signature verification) that limit standard repackaging attacks, some fake and malicious versions of the WhatsApp messenger app were distributed in the wild [6]. These malicious apps were downloaded more than a million times from the Google Play Store, and they were able to retrieve sensitive information through the malicious container.

C. MARVEL

In this section, we introduce the MARVEL methodology and how it is implemented, highlighting the key aspects related to the testing phase. We refer interested readers to [26] for a detailed description of our 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 following security 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.

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 system service that does not require root privileges, like Google Play Services (GPS) [11]: 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 (TC). Mobile vendors have already adopted a similar approach. For instance, Samsung Knox [27] 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 *Interconnected AT (IAT)*. The former allows removing portions of code from the original app, replacing it with fake code (or methods). Thus, it introduces a mitigation against static analysis inspection (i.e., requirement $R1$). To do so, we injected a custom ClassLoader (CCL) into the process of the plugin, which is in charge of i) resolving the requested class, ii) checking if it contains some fake methods, and iii) in case of fake methods, replace them with the correct ones. The latter involves injection controls, which are evaluated during the interaction between the container and the plugin. Thus, the controls fail if the plugin is not executed in the expected container (i.e., requirement $R2$), or the container cannot communicate with the intended plugin (i.e., requirement $R3$). 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

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

attacker is forced to execute and reconstruct the entire package before trying to locate and deactivate all the IAT controls. It is worth noting that, while the MARVEL protection focuses on the Java layer, it is always applicable even to apps entirely written in C/C++: to interact with the native code, an app needs a Java "stub" class with the task to load and interact with C/C++ code. Moreover, the solely Android component entirely supported by the native code is the Activity: an app that wants to interact with a different component has to implement it in a Java layer.

We implemented the MARVEL protection scheme in a prototype tool, called MARVELoid, which injects the protection in the plugins, and we extended the VirtualApp framework [16] to develop the trusted container responsible for the enforcement of the runtime protection. The source code of MARVELoid and the trusted container are available at https://github.com/totoR13/MARVEL.

MARVELoid is a Java-based tool devoted to the generation of the protected plugin starting from the original app: it performs the code splitting and injects the IATs controls (both the base ones and those with encryption). The tool leverages the Soot framework [12] to analyze and modify the bytecode of the Android app, and Jarsigner [22] to sign the output APK file with a valid certificate. MARVELoid requires as input the APK file of the plugin app to protect three input parameters, namely the probability to i) replace a method through code splitting (P_{repl}), ii) inject a base IAT control (P_{base_IAT}), and iii) inject an encrypted IAT (P_{enc_IAT}), and the Package Name (PN) that identifies the part of the app (i.e., the group of classes) to include in the protection process. At the end of the protection process, MARVELoid recreates the `classes.dex` file(s), builds the APK and re-signs it with a valid certificate. Finally, MARVELoid extracts the metadata information of the resulting APK required by the container to perform integrity checks on the plugin. On the other hand, the TC is an extension of the VirtualApp framework in charge of offering the virtualized environment to execute the plugins and enforcing the MARVEL protection at runtime. In detail, the container i) performs the integrity checks of the plugin, ii) interacts with the plugin through the IATs, and iii) restores the original content of the fake methods (if the check succeeded).

III. Testing Methodology

In this section, we introduce the main goals of our experimental campaing, and we detail the stages of our testing pipeline.

A. Testing Goals

The experimental campaign aims to verify the applicability of MARVEL protection scheme in the wild and the fulfillment against the repackaging attacks. To do so, we advocate that the testing pipeline should achieve the following testing goals (*TG*):

a) ***TG1*** *— Correctness:* The pipeline evaluates the MARVEL protection to find possible errors (i.e., failures during the protection process or the runtime evaluation), improve the quality of the software, as well as verify and validate the transformation done. In our scenario, we verify how many apps can i) be statically protected with MARVELoid, and ii) be executed in the Trusted Container. In addition, we investigate the failure samples to identify the causes of such exceptions. For instance, MARVELoid modifies the bytecode of the plugin app (e.g., performs the code splitting), and we have to verify if such modifications result in a valid DEX file.

b) ***TG2*** *— Performances:* The pipeline aims to evaluate how the tools (i.e., MARVELoid and Trusted Container) perform in terms of time and stability under a particular *workload*, which is the combination of plugin app and input parameters applied on it and affecting the number of injected protections.

On the one hand, we aim to evaluate the time for the MARVELoid transformation and the artifacts introduced by the protection mechanism (i.e., size overhead between the protected and original versions of an app). On the other hand, it is important to verify the feasibility of the MARVEL protection at runtime, monitoring the resource usage (e.g., CPU and memory) during the app execution to identify the overhead introduced by our protection scheme.

c) ***TG3*** *— Repeatability:* The principle of *repeatability*, also known as *replicability* and *reproducibility*, refers to the possibility to obtain the same results with a high degree of reliability when the experiments are replicated. In our scenario, the testing pipeline aims to perform the same analysis, despite the set of involved apps.

In addition, the testing pipeline should also guarantee the *runtime reproducibility*, which refers to the amount of resources used by the different versions of the same app (e.g., protected vs. original). To do so, the pipeline executes the same sequence of graphical input events for all versions of the same app, computing the overhead for each click. Thus, different executions of the same app lead to similar results, and the computed runtime overheads are reliable (because reproducible).

As an illustrative example, we can consider an app with two buttons: the first one (`A`) performs an easy calculation (e.g., increment of a number by 1), while the second (`B`) does a heavy computation (e.g., the factorial of a number). If the overhead of resources is computed after clicking the button `A` of the original app and after clicking button `B` of the protected version of the same app, the results would be strongly influenced by the semantics of the app (i.e., heavy vs. light computation). Thus, the pipeline needs to guarantee the runtime reproducibility by emulating the identical sequence of input events.

d) ***TG4*** *— Effectiveness:* MARVEL is an anti-repackaging protection scheme, which aims to protect an Android app from both standard and virtualization-based repackaging attacks. Since a protection scheme is considered effective only if it is harder to circumvent it than to reimplement a malicious app from scratch [21], the security evaluation of MARVEL plays a crucial role in the testing pipeline. In particular, we need i) to evaluate how MARVEL is reliable against different reverse-engineering techniques, and ii) to identify the steps involved in the repackaging process to overcome our protection.

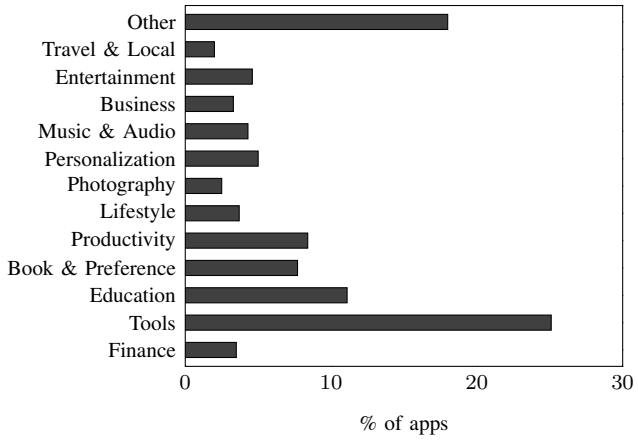
It is worth noticing that our testing pipeline takes into consideration that MARVELoid accepts three input parameters (i.e., P_{repl}, P_{base_IAT}, and P_{enc_IAT}), which may affect the

number of injected protections. For instance, we evaluate how many failures are related to a specific protection mechanism (e.g., code splitting) or how much the protection time of MARVELoid varies depending on those parameters.

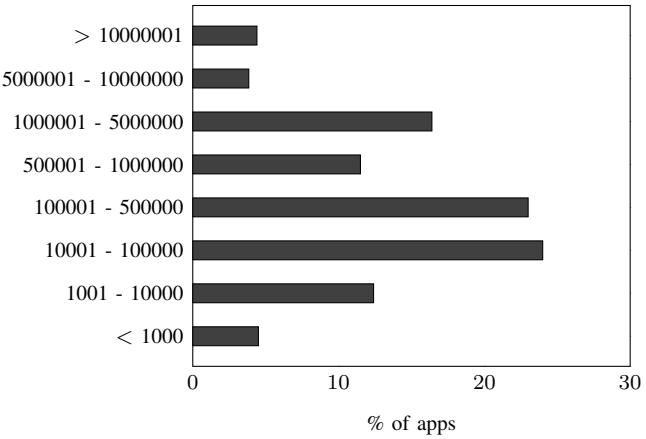
B. Dataset

We need to evaluate our protection scheme against several real-world apps, which are widely used by Android end-users. To do so, we built a comprehensive and diversified dataset of 4000 Android apps collected from the Google Play Store in December 2020. Regarding the dynamic analysis, we randomly selected a subset of 45 apps (cf. Section V). To do so, we extracted useful app information (e.g., package name and number of downloads) using Google Play Scraper [1], and we downloaded the samples with the Playstore Downloader [2].

Figure 2 shows the distribution of the apps category (2a) and the number of their downloads (2b). The most representative category is *Tools* and the *Other* category is a collection of several app families (e.g., *Communication* and *Game*) which accounts for less than 2% of apps each. It is worth noticing that the apps belong to 43 different categories, and most of them (i.e., 87.3%) have been downloaded between 1000 and 5000000 times. Furthermore, the apps of the dataset have an average rating of 4 stars on the Google Play Store, and 60% of them have the `minSDK` above Android 4.0 to include the most recent API features.



(a) Distribution of the apps category.



(b) Distribution of the apps download.

Fig. 2: Distribution of categories and download of the 4000 apps.

IV. MARVEL Testing Pipeline

In this section, we perform an in-depth analysis of how our testing pipeline fulfills the testing goals for each phase introduced in Section III-A.

As mentioned in Section II-C, MARVEL is implemented through two components: MARVELoid is in charge to protect a plugin app by statically injecting different protection mechanisms, while the TC is an Android app that extends the VirtualApp framework to enforce the protection at runtime. As a result of our MARVEL design, we divided the testing pipeline in two stages: *static* and *dynamic analysis*. The first — *static analysis* — aims to verify the reliability and efficiency of the transformation process performed by MARVELoid according to the different protection levels achieved depending on the input parameter values. *Dynamic analysis*, instead, tests the MARVEL protection at runtime by evaluating its effectiveness and usability. In particular, the pipeline executes the protected plugin apps in the TC to validate their execution by verifying i) the impact in terms of resource usage by the Android virtualization, IATs, and code splitting techniques, and ii) the robustness of the prototype at runtime.

In the remaining, we discuss the two phases in detail with the support of Figure 3, which depicts the main steps involved in our testing pipeline.

A. Static Analysis

MARVELoid is distributed as a JAR file, and we recall that it requires as input three parameters (i.e., P_{repl}, P_{base_IAT}, and P_{enc_IAT}) and the app to protect[3].

[3]For more details, refer to the official Github repository https://github.com/totoR13/MARVEL

The testing pipeline introduces the protections into the app under test (AUT) with different combinations of the *workloads* (i.e., input parameters) by following a *producer-consumer* pattern: in **Step 1**, the main process saves the pair `(app, workload)` — e.g., `(com.package.name, (`$P_{repl} = 5$`, `$P_{base\_IAT} = 5$`, `$P_{enc\_IAT} = 5$`))` — into a queue, which is queried by several consumer processes. The consumer processes — also known as workers — are in charge to run the MARVELoid tool (**Step 2.a**) and extract the protection results (**Step 2.b**). In our example, the worker protects the `com.package.name` app by setting P_{repl}, P_{base_IAT}, and P_{enc_IAT} parameters equal to 5. Thank to this design, the pipeline satisfies the testing requirement ***TG3***: whenever the main process injects the same workload values, the workers perform the same tests.

In case of successful protection, the workers measure the MARVELoid execution time and the size of the protected APK compared with the original app's ones (***TG2***), and the protection level, i.e., the number of injected protections for each type (***TG4***). Otherwise, the workers extract, parse and store the stack trace of the MARVELoid exception to discriminate the cause of such error (***TG1***). For instance, MARVELoid

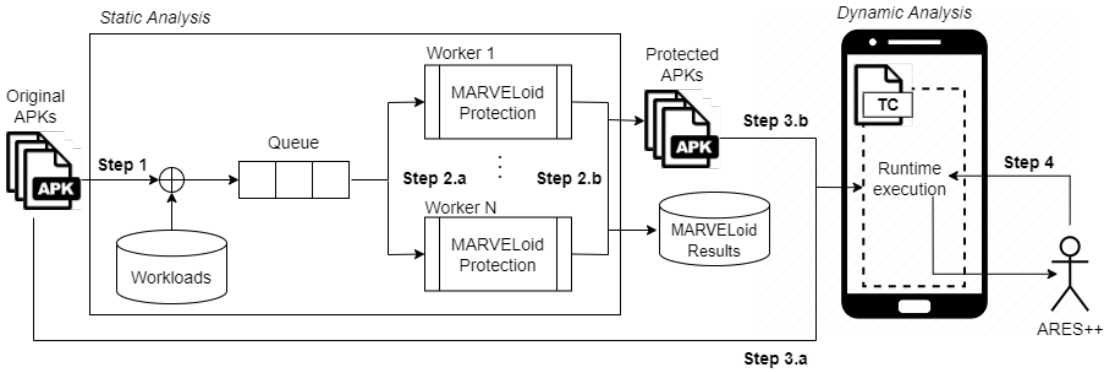


Fig. 3: Testing pipeline.

leverages the `validate` method of the `soot.Body` class in the Soot framework to execute various sanity checks on the modified bytecode. When MARVELoid produces an invalid bytecode, the `validate` throws an exception that specifies which check is failed (e.g., use of an undefined variable).

B. Dynamic Analysis

We recall that the MARVEL protection scheme is built on top of the Android virtualization, leveraging the proxy role of the container (i.e., Trusted Container – *TC*) to prevent repackaging attempts and to enforce the protection at runtime. We divided the runtime evaluation into two phases (respectively, **Step 3.a** and **Step 3.b**) to evaluate the overhead introduced by both Android Virtualization and the MARVEL custom protections (i.e., code splitting and IATs) and their effectiveness.

In the first, the testing pipeline assesses the compatibility of traditional Android apps w.r.t. virtualization and the overhead introduced in terms of CPU and memory usage by the container app. To do so, it uses as a reference values the CPU and memory usage of the traditional Android apps executed directly on the device. In the latter phase, the pipeline evaluates i) the effectiveness of MARVELoid by running the protected apps inside the TC, and ii) the corresponding overhead, using the results of the standard virtualization (previous phase) as reference values. Thus, the dynamic analysis allows us to monitor the overhead introduced by both traditional virtualization and our solution on the set of selected apps. It is important to note that only the apps without any exception in the first phase are protected and executed in the second. This way, we ensure that a failure is attributable to our protection scheme (***TG1***). Moreover, to address ***TG3***, the testing pipeline should be able to reproduce the exact input sequence provided to each app to allow the comparison of resource usage for different app states.

To automatically interact with Android apps (**Step 4**), we modified ARES [25], a black-box tool that leverages Deep Reinforcement Learning. In particular, the extended version of ARES (from now on ARES++) is able to i) execute several plugin apps in a container, ii) ensure the same input sequence for each plugin app, and iii) save the data into a database. The *random* testing strategies stimulate an app under testing by producing a pseudo-random sequence of events, which depends on a *seed* value. Also, we used the *random* algorithm of the ARES tool with a seed that depends on the package name of the app (i.e., the same package name guarantees the same input sequence) to ensure that ARES++ generates the same input sequence. Moreover, ARES considers only the events that trigger a "real" action on the user-interface elements of the app under testing: it generates an event depending on the list of actions that can be carried out in a specific app state [25]. Each time ARES++ generates a new input, it verifies if the app raises a new exception or executes correctly. In the first case, ARES++ parses and stores the output of the `logcat` utility to identify the cause of the fault (***TG1***). Otherwise, it retrieves the CPU and memory usage for the AUT (***TG2***). Since the execution of an app under virtualization is composed of two processes (i.e., container and plugin), the overall CPU and memory usage is given by the sum of the overhead provided by the two processes.

Finally, we manually validate the effectiveness of the MARVEL schema (***TG4***) by impersonating an attacker. In particular, we identify the reverse-engineering techniques and tools that an attacker has to implement to overcome the protection. To ensure that our knowledge of MARVEL did not influence the security evaluation, a Computer Engineering master student helped us in the analysis. He has a generic background of security and performed manual analysis with state of the art reverse-engineering tools [21] of the protected apps to bypass the protection and perform a repackaging attack (as described in Section II-A).

V. Experimental Results

In this section, we summarize the results of the empirical assessment of the MARVEL methodology presented in the original ACSAC paper [26]. Moreover, we discuss some details not present in the original evaluation due to space constraints.

A. Static analysis

The experiments were hosted on a virtual machine running Ubuntu 20.04 with 8 processors and 32GB RAM. We conducted tuning tests to detect the on-average best combinations of input percentages (i.e., P_{repl}, P_{enc_IAT}, and P_{base_IAT} — workload) to ensure a reasonable trade-off between the protection overhead values and the protection level. In particular, we

computed the protection overhead and protection values for 24 different workloads (i.e., the permutations of $[5, 10, 20, 30]$).

The automatic testing pipeline performed 96000 protections (i.e., 4k apps for each permutation) with MARVELoid, which generated a valid APK (Fig. 4) in 97.3% of the cases. The protection of a single app took, on average, 98 seconds (standard deviation of 44 seconds). The remaining 2.7% (i.e., 2624 apps) failed due to well-known bugs of the adopted libraries (such as a Soot bug[4]) or crashes during the tool transformation process. In the last case, the main cause of the error is computable to the code splitting technique (i.e., P_{repl}), which removes some needed instruction.

Figure 4 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%.

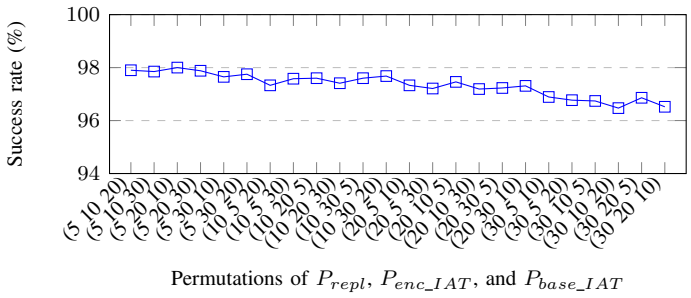


Fig. 4: Percentage of apps successfully protected by MARVELoid.

Finally, MARVELoid was able to inject a minimum of 68 and a maximum of 135 protection elements in the protected apps. All in all, the average percentage of space overhead of the protected APK compared with the original one is always less than 18% (mean of 14.5%, and standard deviation of 1.6).

B. Dynamic analysis

For the testing phase, we used a set of emulated Android 8.0 devices equipped with a dual-core processor and 2GB of RAM.

To evaluate the compatibility with the virtual environment (**Step 3.a** of Figure 3), 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. We recall that, in our prototype, we used Android virtualization as an out-of-the-box tool, leveraging the VirtualApp framework. It is worth noting that such apps are not incompatible with the virtualization per se; in fact, the open-source version of the VirtualApp framework does not support all the newer Android APIs, leading to runtime exceptions when apps implementing these features are executed.

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

Then, in **Step 3.b**, we executed the protected version of the remaining 40 apps into the Trusted Container. In particular, we tested two different combinations of the workloads, i.e., the input parameters are: $P_{repl} = P_{base\_IAT} = P_{enc\_IAT} = 10\%$ ($Setup_{10\%}$) and $P_{repl} = P_{base\_IAT} = P_{enc\_IAT} = 15\%$ ($Setup_{15\%}$). The protected apps contain, 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, with a minimum of 50 protection in both the setups.

In both **Step 3.a** and **Step 3.b**, we executed each app 10 times for 4 minutes to verify its correct execution inside the Android virtualization and the introduced overhead. Moreover, the apps are stimulated with ARES++, an extended version of the ARES tool, which ensures that the sequence of events generated for each app is always the same (cf. Section IV-B).

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[5]. The bug affects the transformation process from the bytecode to the Jimple of third-party libraries included in APK files, causing the final app to crash during the execution, even if its bytecode is valid.
- 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[6]. 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[7] of the app, which mainly ends up in a null pointer exception due to the absence of the correct reference. To solve this issue, we have to include a global reference of the Context object in the Application class. The second bug occurs if a replaced method is already loaded in the custom ClassLoader: a crash can occur because the CCL does not inject the correct body of the method, causing a runtime exception. In such a case, the CCL should prevent the garbage collector from removing the replaced method and restoring the original (fake) method.

Table I shows the minimum, average, and maximum percentage overhead values introduced by the Android virtualization (first column) and our protection mechanisms in

[5]See the related Github issues https://github.com/soot-oss/soot/issues/1151, and https://github.com/soot-oss/soot/issues/1615

[6]https://github.com/appium/appium/issues/12555

[7]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 I: CPU and memory usage overhead in percentage point (pp).

case of the two setups (respectively, $Setup_{10\%}$ in the second and $Setup_{15\%}$ in the third column). The average overhead introduced by the virtualization is negligible in terms of CPU usage (i.e., an increase of 0.9 percentage points), while 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 analysis of the protected apps with $Setup_{15\%}$ confirms the results on the memory overhead. In particular, on average, the memory overhead is negligible, with a peak of 0.7 percentage points in the worst case. On the contrary, the CPU overhead is 4.7 pp, reaching, in the worst case, an increase of 24.5 pp.

Protection Resilience. From a security point of view, MARVEL is effective. With respect to the repackaging attacks presented in Section II-A, 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 becomes more challenging for the attacker due to the code splitting and the IAT with encryption, which removes or encrypts some portions of code from the protected app. Concerning the identification and neutralization of the repackaging protections the victim app might be equipped with, 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. 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. 1) is prevented by MARVEL thanks to the mutual integrity checks between the Trusted Container and the plugin, which stops the execution of the latter in case the environment is detected to be not secure. Overall, MARVEL makes virtualization-based anti-repackaging attacks as complex as traditional ones.

VI. Related Work

In this section, we summarize the related work on repackaging attacks and countermeasures, with a specific focus on the testing methodology.

A. Anti-Repackaging

Anti-repackaging — also known as *repackaging avoidance* or *self-protection* — 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 controls.

The first anti-repackaging technique for mobile apps was proposed in 2015 by Protsenko et al. in [24]. The main idea is to encrypt the *classes.dex* files in the APK and dynamically decrypt them during the app execution. The authors evaluated their technique over a dataset of 749 apps from the F-Droid app store [13], but only 312 apps can be properly protected. The runtime evaluation is performed with Android 4.4.2 devices, and the apps are stimulated through the Android Monkey UI Exerciser framework.

In 2016, Luo et al. in [17] proposed SSN, which injects a set of detection nodes into the app source code (i.e., through the Java bytecode). The viability of the SSN approach has been empirically evaluated on a set of 600 apps belonging to ten app categories in F-Droid, and the experiments were conducted on emulators equipped with Android 4.1. The authors pointed out that all tested apps have been executed successfully by having an average time overhead between 6.4% and 12.4%.

In 2017, Song et al. [30] 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. AppIs has been evaluated on a set of only 8 apps, which are evaluated by multiple Android version emulators (spanning from 4.4 to 6). The results indicate that no app fails, and the space overhead is up to 2% in the worst case, while time overhead spans from 2% to 140%.

In 2018, Chen et al. [7] proposed an SDC scheme, encrypting pieces of code. The ciphered portion of code is decrypted and executed at runtime only if the app is not repackaged. The dataset used for the evaluation is composed of 20000 apps that belong to 15 different categories. The testing phase involved both real users and automatic testing, and the protection introduced a time overhead below 4% for each app tested and a negligible space overhead.

In the same year, Zeng et al. [36] designed BombDroid, a defending technique that leverages logic bomb as anti-repackaging protection for Android apps. The authors built a dataset of 963 taken from F-Droid. They exploited Dynodroid [19] to execute each app for an hour, repeating the experiment 50 times per app. According to the authors' results, the execution time overhead is almost negligible (i.e., $< 2.7\%$), and the space overhead ranges from 8% to 13%.

In 2019, Tanner et al. [31] proposed an extension of BombDroid. The main advancement is implementing the logic bombs (i.e., AT controls and original code) in the native code. The authors tested their solution against 100 apps downloaded

from the Google Play Store and F-Droid and presented that only 47% of apps could be transformed without runtime exceptions. The runtime evaluation was performed on a real device (i.e., LG Nexus 5X), running Android 8.1.0, and each app has been tested through Monkey to generate random user's input: the time overhead is from 1.42% up to 10.42%, while the space overhead is negligible.

In 2020, Merlo et al. [21] defined some guidelines for reliable anti-repackaging techniques and proposed a new anti-repackaging methodology, i.e., ARMAND [20], that leverages a pseudo-stochastic criterion to inject multiple types of logic bombs both in the Java bytecode and in the native code. The evaluation phase of ARMANDroid is performed on a dataset of 30000 real-world Android apps downloaded from the Google Play Store and F-Droid. For the runtime evaluation, the authors used an emulated Android 8.0 phone equipped with the latest version of Google Play Services. Authors perform both Static and Dynamic analysis, showing that 92% (out of 30000) of the app can be successfully protected, and 87% (out of 200) are executed without any runtime exception.

It is important to highlight that previous solutions rarely pay attention to how the dataset is built and how the apps are tested (e.g., differentiating between the protection phase — static analysis — and the runtime evaluation — dynamic analysis). Moreover, only ARMAND takes into consideration the security aspect of the proposed protection scheme (i.e., ***TG4*** *in the MARVEL testing pipeline).*

B. Anti-Virtualization

Malicious exploitation of the virtualization technique [38], [37], [28], [10] has motivated researchers towards the design of possible defense mechanisms [34], [28], [10], [18], [29], which, to some extent, might also defend from virtualization-based repackaging attacks. Most of them (i.e., [34], [28], [33], [18]) are designed to detect whether an app is running in a virtual environment. To achieve such goal, 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). To test such solutions, authors collected several container app (e.g., 7 and 11 container apps respectively for [18] and [10]). Then, they executed the protected apps in the virtual environment and verified how many container apps were detected, and eventually after how much time. For instance, in [18], the authors implemented six different controls. The results show that only two controls are effective against all container apps, while the other failed in some cases.

On the contrary, VAHunt [29] relies on control-flow graphs to see whether a plugin component is eventually replaced by a stub one. If it is detected, VAHunt checks whether there is any stealthy loading of (potentially harmful) code. To test the VAHunt effectiveness, the authors built three datasets: Android app based on virtualization, apps without virtualization, and dual-instance apps. Their evaluation pipeline is divided into two phases to validate each aspect of the detection mechanism. In the former, the authors validated the virtualization detection engine, while the latter aimed to verify the detection accuracy of the code loading strategies.

In contrast with Anti-Repackaging, most testing methodologies for the Anti-Virtualization techniques only validate their effectiveness, leaving the other testing requirements uncovered. Moreover, most datasets are insufficient because they consider only a few apps (e.g., the top ones) and do not cover the heterogeneity of the real-world scenario. In particular, only VAHunt considers different aspects through a testing pipeline evaluated against a comprehensive dataset.

VII. Conclusion, Discussion & Future Work

In this paper, we presented the testing pipeline adopted for MARVEL, a methodology that allows protecting Android apps against both traditional and virtualization-based repackaging attacks. The testing pipeline encompasses a *static analysis* phase that verifies the transformation process done by MARVELoid, and a *dynamic analysis* phase that validates at runtime the protections injected into the Trusted Container and into the plugin apps.

From a technical standpoint, the pipeline can automatically fulfill all the testing requirements (i.e., *correctness*, *performance*, *repeatability*, and *effectiveness*), except for the ***TG4*** during the dynamic analysis. To the best of our knowledge, there is no state-of-the-art technology that can simulate the repackaging attack in the Android ecosystem. Despite the several research work proposed so far [15], [8], [35], the attack simulation is still an open problem. However, several rating systems allow to evaluate the security of a protection scheme, which are based on the concept of risk, such as DREAD [9]. Usually, the risk is the probability that a threat occurs with respect the potential damage to the system/software if an attacker is able to exploit it. These rating systems have a downside that they are quantitative because they are based on a model of the entire system. In our manual security evaluation, in 14 days, the master student was able to automatically detect the splitted methods, but he was not able to remove and bypass the IAT controls. We acknowledge that the analysis is not exhaustive, but it is representative of a mean-skilled attacker.

From an experimental point of view, our evaluation of MARVEL with over 4.000 Android apps demonstrated the applicability and efficacy of the tool and the proposed protection scheme. Moreover, we designed the testing pipeline to ensure the reproducibility of our tests, and we published the result in the GitHub repository https://github.com/totoR13/MARVEL. As a future extension of this work, we plan to improve the exception handling in both Static and Dynamic analysis to perform improved automatic analysis of the MARVEL failures.

REFERENCES

[1] "Google play scraper," https://github.com/facundoolano/google-play-scraper, accessed September 6, 2022.

[2] "Playstore dowloader," https://github.com/ClaudiuGeorgiu/PlaystoreDownloader, accessed September 6, 2022.

[3] "Acsac 2021 artifacts," 2021, accessed online: September 6, 2022. [Online]. Available: https://www.acsac.org/2021/program/artifacts/

[4] M. Alecci, R. Cestaro, M. Conti, K. Kanishka, and E. Losiouk, "Mascara: A novel attack leveraging android virtualization," 2020.

[5] Avast, "Malware posing as dual instance app steals users' twitter credentials," 2016, accessed online: September 6, 2022. [Online]. Available: https://blog.avast.com/malware-posing-as-dual-instance-app-steals-users-twitter-credentials

[6] BBC, "Fake whatsapp app downloaded more than one million times," 2017, accessed online: September 6, 2022. [Online]. Available: https://www.bbc.com/news/technology-41886157

[7] K. Chen, Y. Zhang, and P. Liu, "Leveraging information asymmetry to transform android apps into self-defending code against repackaging attacks," *IEEE Transactions on Mobile Computing*, vol. 17, no. 8, pp. 1879–1893, 2018.

[8] S.-D. Chi, J. S. Park, K.-C. Jung, and J.-S. Lee, "Network security modeling and cyber attack simulation methodology," in *Australasian Conference on Information Security and Privacy*. Springer, 2001, pp. 320–333.

[9] M. Corporation, "Dread threat model," 2010, accessed online: September 6, 2022. [Online]. Available: https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff648644(v=pandp.10)#dread

[10] D. Dai, R. Li, J. Tang, A. Davanian, and H. Yin, "Parallel space traveling: A security analysis of app-level virtualization in android," in *Proceedings of the 25th ACM Symposium on Access Control Models and Technologies*, ser. SACMAT 20. New York, NY, USA: Association for Computing Machinery, 2020, p. 2532. [Online]. Available: https://doi.org/10.1145/3381991.3395608

[11] Google, "Overview of google play services," https://developers.google.com/android/guides/overview, Google, 2021, accessed online: September 6, 2022.

[12] S. R. Group, "Soot - a java optimization framework," 2021, accessed online: September 6, 2022. [Online]. Available: https://github.com/soot-oss/soot

[13] C. Gultnieks, A. A. Fuentes, A. Oberhauser, and A. Demirta, "F-droid," 2020, accessed online: September 6, 2022. [Online]. Available: https://www.f-droid.org/

[14] I. Innovations, "Parallel accounts," 2020, accessed online: September 6, 2022. [Online]. Available: https://play.google.com/store/apps/details?id=com.in.parallel.accounts

[15] M. E. Kuhl, M. Sudit, J. Kistner, and K. Costantini, "Cyber attack modeling and simulation for network security analysis," in *2007 Winter Simulation Conference*. IEEE, 2007, pp. 1180–1188.

[16] J. L. N. T. C. Ltd., "Virtualapp," 2020, accessed online: September 6, 2022. [Online]. Available: https://github.com/asLody/VirtualApp

[17] L. Luo, Y. Fu, D. Wu, S. Zhu, and P. Liu, "Repackage-proofing android apps," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016, pp. 550–561.

[18] T. Luo, C. Zheng, Z. Xu, and X. Ouyang, "Anti-plugin: Don't let your app play as an android plugin," in *Proceedings of Blackhat Asia 2017*, 2017.

[19] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: Association for Computing Machinery, 2013, p. 224234. [Online]. Available: https://doi.org/10.1145/2491411.2491450

[20] A. Merlo, A. Ruggia, L. Sciolla, and L. Verderame, "Armand: Anti-repackaging through multi-pattern anti-tampering based on native detection," *Pervasive and Mobile Computing*, vol. 76, p. 101443, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1574119221000857

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

[22] Oracle, "Jarsigner," Oracle, 2021, accessed online: September 6, 2022. [Online]. Available: https://docs.oracle.com/javase/7/docs/technotes/tools/windows/jarsigner.html

[23] PAGalaxyLab, "Yahfa," 2021, accessed online: September 6, 2022. [Online]. Available: https://github.com/PAGalaxyLab/YAHFA

[24] M. Protsenko, S. Kreuter, and T. Mller, "Dynamic self-protection and tamperproofing for android apps using native code," in *2015 10th International Conference on Availability, Reliability and Security*, 2015, pp. 129–138.

[25] A. Romdhana, A. Merlo, M. Ceccato, and P. Tonella, "Deep reinforcement learning for black-box testing of android apps," 2021.

[26] A. Ruggia, E. Losiouk, L. Verderame, M. Conti, and A. Merlo, "Repack me if you can: An anti-repackaging solution based on android virtualization," in *Annual Computer Security Applications Conference*, ser. ACSAC. Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3485832.3488021

[27] Samsung, "Samsung knox," 2021, accessed online: September 6, 2022. [Online]. Available: https://www.samsungknox.com/en

[28] L. Shi, J. Fu, Z. Guo, and J. Ming, "jekyll and hyde is risky: Shared-everything threat mitigation in dual-instance apps," in *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys 19. New York, NY, USA: Association for Computing Machinery, 2019, p. 222235. [Online]. Available: https://doi.org/10.1145/3307334.3326072

[29] L. Shi, J. Ming, J. Fu, G. Peng, D. Xu, K. Gao, and X. Pan, "Vahunt: Warding off new repackaged android malware in app-virtualization's clothing," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 535549. [Online]. Available: https://doi.org/10.1145/3372297.3423341

[30] L. Song, Z. Tang, Z. Li, X. Gong, X. Chen, D. Fang, and Z. Wang, "Appis: Protect android apps against runtime repackaging attacks," in *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2017, pp. 25–32.

[31] S. Tanner, I. Vogels, and R. Wattenhofer, "Protecting android apps from repackaging using native code," in *Foundations and Practice of Security*, A. Benzekri, M. Barbeau, G. Gong, R. Laborde, and J. Garcia-Alfaro, Eds. Cham: Springer International Publishing, 2020, pp. 189–204.

[32] M. Team, "Multiple accounts:parallel app," 2021, accessed online: September 6, 2022. [Online]. Available: https://play.google.com/store/apps/details?id=com.excelliance.multiaccounts

[33] L. Tech, "Parallel space - fmulti account," 2021, accessed online: September 6, 2022. [Online]. Available: https://play.google.com/store/apps/details?id=com.lbe.parallel.intl

[34] Y. Wu, J. Huang, B. Liang, and W. Shi, "Do not jail my app: Detecting the android plugin environments by time lag contradiction," *Journal of Computer Security*, vol. 28, pp. 1–25, 01 2020.

[35] R. V. Yohanandhan, R. M. Elavarasan, P. Manoharan, and L. Mihet-Popa, "Cyber-physical power system (cpps): A review on modeling, simulation, and analysis with cyber security applications," *IEEE Access*, vol. 8, pp. 151 019–151 064, 2020.

[36] Q. Zeng, L. Luo, Z. Qian, X. Du, and Z. Li, "Resilient decentralized android application repackaging detection using logic bombs," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, ser. CGO 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 5061. [Online]. Available: https://doi.org/10.1145/3168820

[37] L. Zhang, Z. Yang, Y. He, M. Li, S. Yang, M. Yang, Y. Zhang, and Z. Qian, "App in the middle: Demystify application virtualization in android and its security threats," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 3, no. 1, Mar. 2019. [Online]. Available: https://doi.org/10.1145/3322205.3311088

[38] C. Zheng, T. Luo, Z. Xu, W. Hu, and X. Ouyang, "Android plugin becomes a catastrophe to android ecosystem," in *Proceedings of the First Workshop on Radical and Experiential Security*, ser. RESEC '18. New York, NY, USA: ACM, 2018, pp. 61–64. [Online]. Available: http://doi.acm.org/10.1145/3203422.3203425