Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Master Degree in Cybersecurity

Virtualization-Based Malwares: Can We Defend Against Them?

Supervisor
Prof. Eleonora Losiouk
Università degli Studi di Padova

Master Candidate
Simone Zerbini

Student ID
2026829

Academic Year
2021-2022

Abstract

App-Virtualization is a technique that allows an application, called host or container, to create a virtual environment on top of the Android framework. In this virtual environment, other applications, called plugins, can be executed from their apk without being installed on the device.

This technique can be used to offer some interesting features, but it can also be exploited for malicious purposes. For instance, it can be exploited to evade anti-malware detection by dynamically loading malicious code. Another common malicious use is to simplify the repackaging of an application: with the standard approach, an attacker must decompile the apk of the target application and then add the malicious payload before he can distribute the repackaged app, on the other hand, by exploiting virtualization it is enough to execute the target application as a plugin in a malicious container.

Currently, the countermeasures at our disposal are Third-party Anti-Malware, Anti-Plugin techniques and the state-of-the-art tool VAHunt. Anti-Plugin techniques refer to a series of methods that a developer can implement in his application to ensure that it does not run in a virtual environment. Unfortunately, most of these techniques can be easily bypassed, but the major limitation is that they are rarely adopted by developers. VAHunt is a tool to check whether an app makes use of virtualization, additionally it is able to detect certain suspicious uses of the latter. It has been observed that this tool has some flaws. These are mainly due to the fact that it was built considering just the 2 main virtualization frameworks, namely VirtualApp and DroidPlugin.

In this thesis, the behaviour of these malware was investigated in more detail. In particular, malwares were analyzed through both static and dynamic reverse engineering techniques. In addition, it has been proposed Matrioska, a new tool that exploits app-virtualisation itself to perform online a dynamic analysis of applications. This tool is able to detect the malicious use of app-virtualization as an alternative to repackaging with close to 100% accuracy.

Contents

Listing of figures

Listing of tables

Listing of acronyms

APK . . . . . Android Package

DEX . . . . . Dalvik Executable File

1 Introduction

App-Virtualization is a technique that allows an application, called host or container, to create a virtual environment on top of the Android framework. In this virtual environment, other applications, called plugins, can be executed from their apk without being installed on the device. This technique can be used to offer some interesting features, for example, it makes it possible to run two instances of the same app on the same device simultaneously, or it can promote modular programming in which certain functions of an app are downloaded only if and when a user requests them, thus saving memory.

Unfortunately, it can also be exploited for malicious purposes. For instance, it can be exploited to evade anti-malware detection by dynamically loading malicious code. Another common malicious use is to simplify the repackaging of an application: with the standard approach, an attacker must decompile the apk of the target application and then add the malicious payload before he can distribute the repackaged app, on the other hand, by exploiting virtualization it is enough to execute the target application as a plugin in a malicious container. These and other malicious uses of app-virtualization have been previously pointed out by [2][1][3][4][5].

Currently, to counter these malware we can rely on 3 different approaches. The first defensive layer at our disposal includes google play protect and other third-party anti-malware with a cloud-based architecture. In fact, anti-malware with this architecture are able to analyze an

application with very powerful means and rely on features obtained both through static and dynamic analysis, thus having the potential to detect also malware based on app-virtualization. The second approach is Anti-Plugin techniques [2][1][3], these refer to a series of methods that a developer can implement in his application to ensure that it does not run in a virtual environment. Unfortunately, most of these techniques can be easily bypassed, but the major limitation is that they are rarely adopted by developers. Finally there is VAHunt [6], which is an offline tool based on static analysis and currently it represents the state-of-the-art. This tool can check whether an app makes use of virtualization, additionally it is able to detect certain suspicious uses of the latter. It has been observed that this tool has some flaws. These are mainly due to the fact that it was built considering just the 2 main virtualization frameworks, namely VirtualApp and DroidPlugin.

In this thesis, the behaviour of these malware was investigated in more detail and the effectiveness of existing countermeasures was tested. In particular, malwares were analyzed through both static and dynamic reverse engineering techniques. In addition, it has been proposed a new tool called Matrioska. Our is an online tool, i.e. it can operate directly on the user's device It can detect the malicious use of app-virtualization as an alternative to repackaging, overcoming some of the VAHunt's flaws. To do this, dynamic analysis is exploited by executing the sample in a virtual environment, so, if the sample implements virtualization, we will have a plugin executed in the virtual environment of the sample which, in turn, will be executed in the virtual environment of our tool, for this reason the tool has been called Matrioska.

This approach has been proposed before. In fact, in [7] and [8] are proposed two sandboxing tools that run on Android systems, called NJAS and Boxify respectively. These tools are able to run in their own context and instrument other applications, without requiring changes to the Android framework or special permissions.

The rest of the thesis is organised as follows. In Chapter 2, we provide background on the topic, in particular we explain how app-virtualization works and how it is exploited for malicious purposes. Existing countermeasures are described in Chapter 3, in particular cloud-based protection, anti-plugin techniques and VAHunt. In Chapter 4, System Model, Threat Model and Solution Requirements are presented. Chapter 5 describes how Matrioska works and our design choices. Some implementation details are described in Chapter 6. Chapter 7 reports the results obtained with Matrioska, moreover some of the most representative malware we anal-

ysed are described. In Chapter 8 we discuss the limitations of our approach and how this work will be carried forward in the future. Finally, Chapter 9 concludes the thesis.

2

Background

2.1 How App-Virtualization Works

The basic concept behind virtualization is a proxy hook component. The proxy must be capable of intercepting the API and function calls of the plugin application, modify the parameters, and forward them to the Android system. Then it has to do the same the other way around, that is intercepting the Android system responses and re-elaborating them in order to forward them to the plugin application.

But this is not enough, in order to offer a virtual environment the host has to face 3 main challenges: execute the code of a not-installed apk, manage the lifecycle of the plugin components and manage memory accesses.

To overcome the first challenge a ClassLoader hook is used. In Android system, APK or dex files are loaded by ClassLoader. The loaded files and classes are stored in a list defined in the ClassLoader object called DexElement. Whenever the ClassLoader needs to load a class, it will scan through this list to match the given classname. If failed to locate in the current ClassLoader, it may trace back to the parent ClassLoader iteratively. There are several types of ClassLoader in Android: BootClassLoader is used to load the system class; PathClassLoader is used to load app class. But all of them are derived from the base class BaseDexClassLoader.

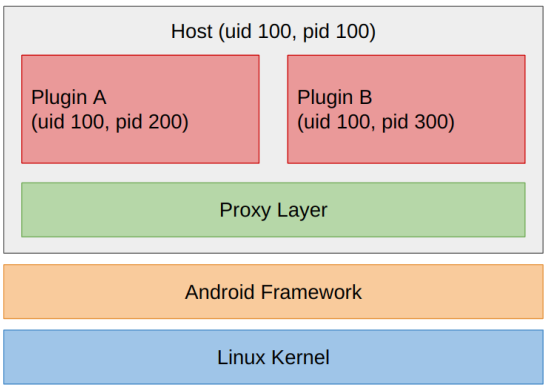

Figure 2.1: App-virtualization architecture.

Originally, system goes to a specific path, usually is 'data/app' folder, to find the installed app's APK file and related resources. At the launching step, only the APK file of the host app is parsed and saved in DexElement list. Since the plugin APK file is not under that specific path, it cannot be automatically loaded by the system. Therefore, if system attempts to use current ClassLoader to load a class defined in plugin file, nothing can be located and the plugin cannot be launched. DroidPlugin will hook this ClassLoader and insert a parsed plugin APK file to the DexElemement list of it. Once this hooking step is done, the classes defined in the plugin are available to search and launch in the normal way.

The host must also manage the lifecycle of the plugin app components. App components, such as Activity, Service, Broadcast Receiver, Content Receiver, are the basic blocks of a mobile app, and only the Android system can manage their lifecycle. First of all, if we try to initialize a plugin app component we will get an error as it is not registered in the host manifest. The host cannot know in advance how many and how are called the plugin components, so to solve this problem the host pre-defines several stub components. A similar approach is used also for permissions: the host app must declare all the permissions required by the plugin for it to work correctly, not being able to know what they are in advance, all permissions are declared. However, these stub components are not enough. In order to initialize a component, an app must communicate with the android system using intents. Thus, in order for these intents to be handled properly, the host must intercept and enclose them within new intents that refer to stub components, so as to fool the Android system. This is called intent wrap operation. Finally, the host has to manage plugin memory. Normally an app has a dedicated memory area

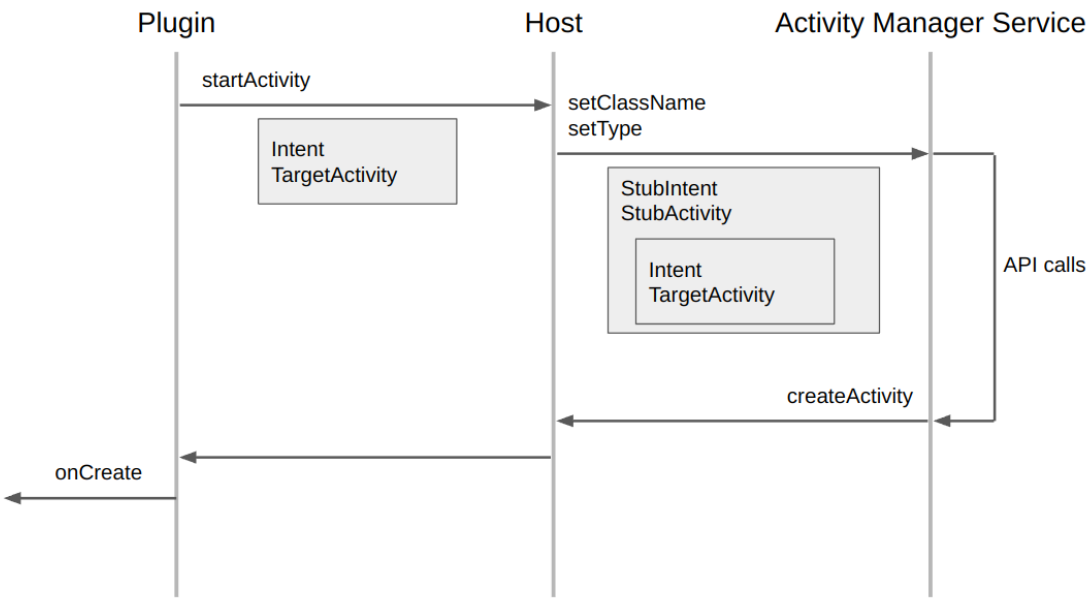

Figure 2.2: Intent wrap operation.

in which to manipulate files, this is located in `/data/user/0/package-name/files`. Since the plugin is not installed on the device, this memory area does not exist and therefore the host has to compensate by redirecting all memory accesses to a directory similar to: `/data/user/0/host-packagename/virtual/data/user/0/plugin-package-name/files`

However, a Virtual Environment built in this way has a problem. Typically, each app installed on the Android system is assigned a unique UID according to its package name. Users cannot install two copies of the same APK file on a single device because of the UID restriction. A plugin app, on the other hand, even if in a different process runs inside the host app, thus sharing the same UID. As a result, different plugins also have the same list of permissions with the host app, and each plugin can access the data of the host or other plugins.

2.2 Malicious Usage

Unfortunately, app-virtualization can be exploited for malicious purposes. Moreover the trend of malicious apps adopting this technique is increasing, as you can see in Figure 2.3. In this section, we describe many different scenarios in which app-virtualization can be exploited for malicious purposes.

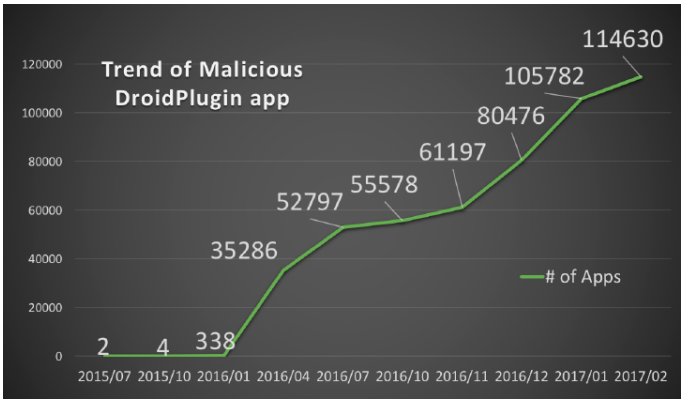

Figure 2.3: DroidPlugin framework abuse trend. [1].

2.2.1 Malicious Plugin

In this scenario, a malicious plugin app exploits app-virtualization vulnerabilities to carry out the attack. These vulnerabilities mainly concern the shared-UID architecture, where hosts and plugins share the same UID.

A first example concerns the abuse of stub permissions. A malicious plugin can in fact appear harmless by not declaring any permissions in its manifest, but once executed as a plugin, it can exploit all the permissions guaranteed by the container.

A second attack of this kind is due to the fact that the isolated sandbox principle implemented by the android system is not respected in the virtual environment. Thus, a plugin app can easily access sensitive data of other plugin apps.

This category has not been explored in depth in this work, as malware of this type does not implement virtualization itself.

2.2.2 Update

In this scenario, malware inserts malicious code within a plugin to be executed in the virtual environment. Once the attacker has tricked the user into installing one of his malware apps on the device, he can update existing malware or install new ones by simply downloading the new apk file from the remote server and replacing the old one. By doing so, the attacker can update the malware without relying on the app store and significantly reducing the chance of being detected by users, since the entire process does not involve user interaction. This behavior has been observed in the PluginPhantom malware [9], which is capable of capturing screenshots, recording audio, intercepting and sending SMS. The uniqueness of this malware is that each of the malicious functionalities is organized as an individual plugin that the host can dynamically load to perform the attack and keep updated.

2.2.3 Evasion

Malwares in this category exploit app-virtualization to load malicious code at runtime in order to evade static analysis, achieving a behavior very similar to other dynamic code loading strategies. To make this technique even more effective the plugin apk file can be encrypted or downloaded when needed.

2.2.4 Alternative to Repackaging

Malware in this category exploits app-virtualization as a simpler and more effective alternative to repackaging. Without the use of virtualization a repackaging attack consists of 3 phases: download of the target app, injection of malicious code, redeployment of the repackaged app. The second phase is different for each app and can be very demanding. In fact, an attacker must first analyze the target app to understand where and how to inject his code, in addition, the target app may contain some repackaging countermeasures (such as encrypted dex [10] and BombDroid [11]) that the attacker must detect and neutralize. Virtualization allows performing a similar attack with a methodology that is always the same and that does not require analyzing and manipulating the target app, by doing so all repackaging countermeasures are also bypassed. In fact, with virtualization it is enough to run the target app as a plugin and inserting the malicious code in the host or in a second plugin. In this way, a phishing attack can

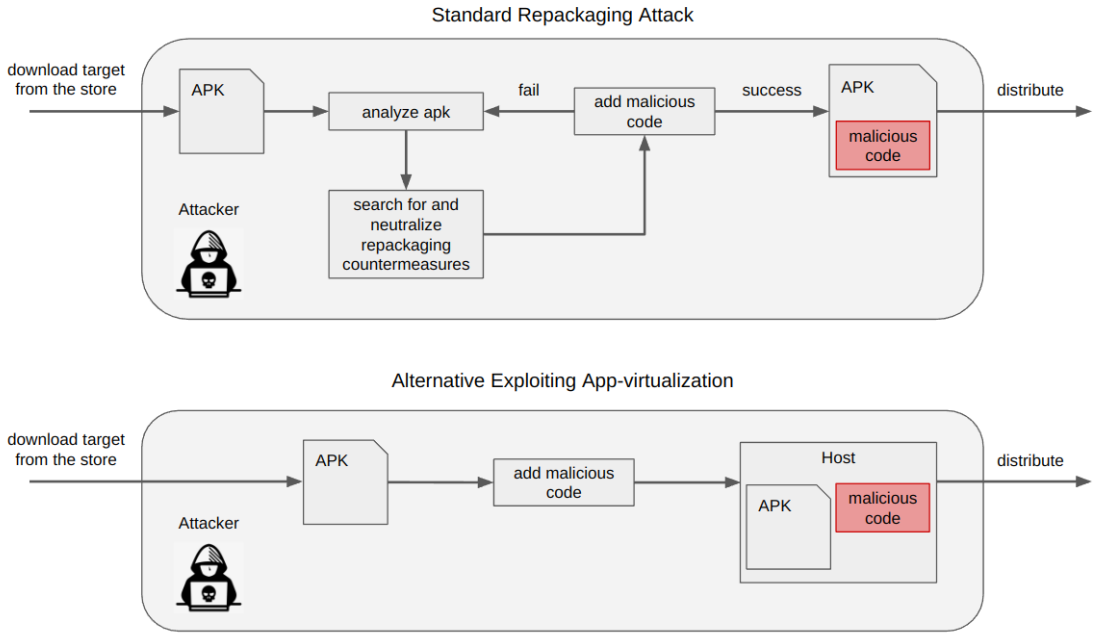

Figure 2.4: Repackaging and its alternative in comparison.

be launched by intercepting user's input or, more simply, the target app can become a carrier for spreading the malicious code.

2.2.5 Adware

Common practice among adware is to download apps and then frequently try to install them, thus promoting the app very aggressively. The limitation in this behaviour is that the user's authorisation is required to install a new application in the Android system. App-virtualization can be exploited to overcome this limitation as no permission is required to install an apk in the virtual environment. Such adware can therefore download and install apps repeatedly and then run them without the user's consent or just create a shortcut on the device's homescreen. On top of subjecting the user to a very aggressive advertisement, this practice is also very dangerous. In fact, several apps unknown to the user will be installed, which will have all permissions declared by the adware, typically all permissions so that any app can be run as a plugin and so promoted.

2.2.6 App confusion

Finally we have the app confusion attack, which is probably the most complex app-virtualization-based attack. Malwares in this category stop the execution of an app and resume it within their virtual environment, then proceed with a phishing attack. The tricky part is being able to resume the target app in the virtual environment without the user noticing the change. At that point the malware has full control over the target app and can then steal user input or other sensitive data.

3

Related Work

3.1 Cloud-based Protection

The trend followed by the major anti-malware for smartphones is to adopt a cloud-based design. This is indeed the design adopted for Google Play Protect and the anti-malware we have considered (AVG, Avast, Bitdefender).

Lets now see what cloud-based design consists of using Google Play Protect (GPP) as an example [12]. GPP analyses each app when it is uploaded to the Play store and, in order to protect the user also from apps from other sources, it analyses apps from other stores, collected by crawling the Internet or even collected by users. The detection mechanism is composed of various techniques such as machine learning and even human analysis, and exploits features extracted by both static and dynamic analysis. In addition, GPP takes advantage of non-code features to determine possible relationships between applications and to evaluate whether the developer that created the application has been associated with the creation of other malware. Finally, GPP analyzes each app when it is installed and also scans the entire device every day for apps classified as dangerous. When one is found, the user is informed with a message suggesting to uninstall the suspicious app, or it is deleted immediately depending on the threat level.

The big advantage of this design is the ability to analyze an application with tools and tech-

niques that could not be used on a device. On the other hand, these anti-malware can't protect against malware that hasn't been scanned before.

These tools are not specifically designed to detect app-virtualization-based malware but, they can analyze applications by powerful means relying on features obtained by both dynamic and static techniques, thus having the potential to detect app-virtualization-based malware as well.

3.2 Anti-Plugin Techniques

These techniques refer to a series of libraries (as DiPrint [2] and PluginKiller [1]) that a developer can integrate into his application, so that it can recognise if it is being executed in a virtual environment and thus limit its functionality or stop its execution. In this way the application will no longer be vulnerable to malicious plugin attacks and will no longer be a target for alternative repackaging attacks. The proposed solutions rely on the following signatures to determine if the application is in an virtual environment.

- As we have seen, the host usually declares most of the Android permissions so, we can either leverage PackageManager or try to access certain restricted resources that requires a permission not declared by the plugin, in order to check if the granted permissions match the permissions declared by the plugin app.

- Since the plugin app has never been installed, it can check whether its package name is registered to the system.

- If we run a service and then we use the API getRunningServices of ActivityManager to get the information of the running service, we will get the name of the stub service used by the host to fool the Android system.

- Host runs in a different process than the plugin but shares the same UID. So we can check the number of processes assigned to our UID and check for unexpected processes.

- Host will parse the manifest file to retrieve the broadcast receivers defined by the plugin, and generate the code to dynamically register them at runtime. We can define a static broadcast receiver in plugin's manifest file, and unregister all of the dynamic receivers at

runtime. In the real Android environment, the static receiver is still effective; but in the virtual environment all of the static and dynamic receivers are gone.

- We can define a broadcast receiver in the plugin's manifest, and set the enabled property of it as false by default. Then, we can use the API setComponentEnabledSetting to enable this receiver at runtime. If it is running as a plugin, this runtime enabling will not take any effect. We can easily send a broadcast that matching this receiver to check whether it is enabled or not, and find evidence of whether the app is running as a plugin.
- Every app has internal storage whose directory is specified by the package name of the app itself. However, the internal storage assigned to a plugin will be a subdirectory of the one assigned to the host, and thus we will have a different path.
- If we are in a virtual environment we can find two different "base.apk" files in process memory by reading "/proc/self/maps". One belongs to the plugin, and the other to the host.
- To detect the involvement of a host app we can throw exception in the components' lifecycle functions and then analyze the related stack traces.
- Some native components share the internal information among instances within a same app. Since host and plugins share the same UID, we can look for information not belonging to the app.
- We can search "/proc/[pid]/maps" file to find the suspicious library that could provide native hooking.

Unfortunately, the vast majority of these signatures can be bypassed, additionally these techniques are rarely adopted by developers.

3.3 VAHunt

VAHunt is an offline static analysis tool aimed at identifying the presence of app-virtualization within an app and its malicious use as alternative to repackaging. This tool was published in 2020 and currently represents the state of the art but, as we will see, it has some shortcomings.

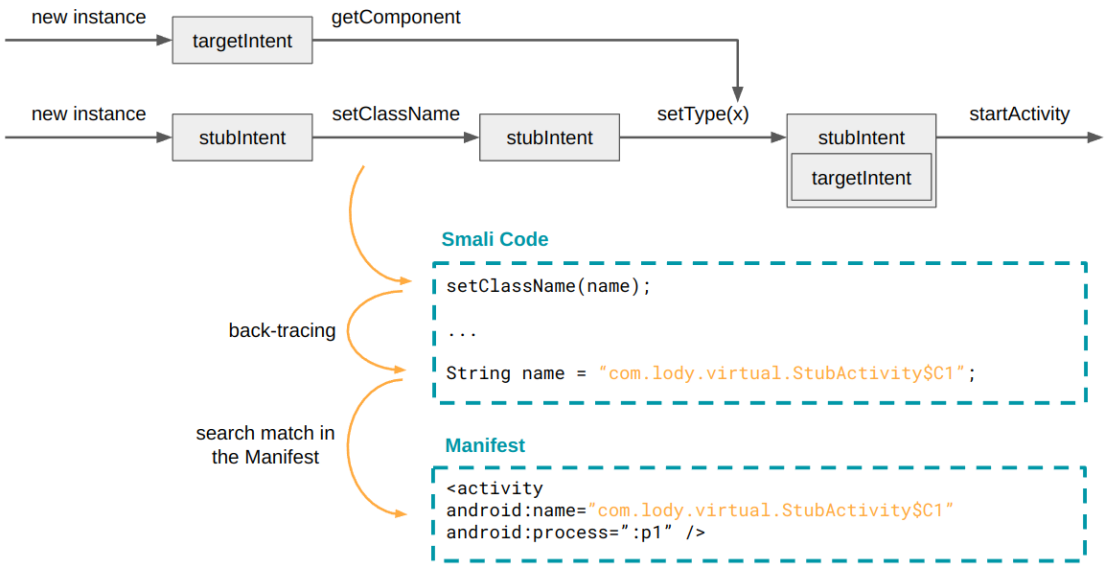

Figure 3.1: VAHunt virtualization engine detection schema.

VAHunt is divided into two parts: the first identifies the presence of an app-virtualization engine, while the second identifies its malicious use. The first part is based on detecting the intent warp operation. This operation must be done by every app-virtualization engine since, in order for the Android system to launch a plugin component, it must be tricked with a stub component. First of all, VAHunt extracts from the smali code all the intent objects and the operations related to them (such as setClassName(), setComponent(), setType(), getComponent(), ...). Once a picture of how each intent is manipulated has been created, VAHunt checks whether this matches with the reference pattern that corresponds to the creation of a stubIntent for app-virtualization. In Figure 3.1 is shown an example of this pattern: it starts with two source Intent creations (e.g., targetIntent and stubIntent); the type of the stubIntent comes from the targetIntent; the stubIntent sets its Class attribute using setClassName() or setComponent() and its final operation is startActivity(). In addition, once a stubIntent has been identified, VAHunt tries to reconstruct the name of the corresponding stub component by tracing back the parameters passed to methods such as setComponent() or setClassName(). Finally, once the name is retrieved, it looks for a match between the predefined components in the manifest, so as to find confirmation that it is a stub component.

The second part, aimed at recognizing suspicious behavior related to app-virtualization, is

instead based on the identification of what is called silent loading. Silent loading refers to the behavior whereby an host app installs and executes an apk in the virtual environment without user interaction. This behavior is very common in app-virtualization-based malware but is required for malware that uses app-virtualization as an alternative to repackaging. Indeed, a malware of this kind must install and run the plugin immediately at startup so as to be able to deceive the victim. An application may contain a app-virtualization frameworks without ever making use of them, so as a first step VAHunt looks for APIs to access the most common memory areas where the apk of a potential plugin may be stored (such as getFilesDir(), getExternalStorageDirectory() and getAsset()), and checks whether these APIs are used to pass objects to the installation interfaces offered by app-virtualization frameworks. Even if the plugin is downloaded at runtime it will need to be stored somewhere at some point. Once one of these installation methods has been found, VAHunt traces the method calls backwards to check whether they are called as a consequence of user interaction (i.e. as a consequence of methods such as onClick()) or whether they are called in component lifecycle methods (such as onCreate()). If we find ourselves in the second case, i.e. a silent loading behavior has been identified, the sample is tagged as malware.

As anticipated, we have noticed some limitations in VAHunt. The first is that VAHunt is a static analysis tool which analyzes the smali code but it cannot analyze native code and it is also vulnerable to obfuscation. This is a big limitation since obfuscation is a widely used technique in application development and even more when it comes to malware.

The second major limitation of VAHunt is the fact that it is heavily specialized on VirtualApps and DroidPlugins. This can be mainly noticed in two points:

- To locate an intent wrap operation, VAHunt looks for a match with only some very specific patterns designed on the two frameworks.
- The installation interfaces used to detect a silent loading behavior are specific methods of classes belonging to the 2 frameworks.

Still regarding the detection of a silent loading behavior, it can be noted that VAHunt searches only for the installation operation of a plugin and not for execution. This can lead to misclassification of an app that installs a plugin at start-up but without executing it.

Lastly, we found several flaws in the code that allow an attacker to bypass VAHunt by applying only minor modifications to the two frameworks VirtualApp and DroidPlugin.

4

Assumptions and Requirements

4.1 System Model

The victim is a generic Android user, who regularly downloads applications from the store and who owns a device with an official Android release without any root privileges.

4.2 Threat Model

In our scenario, the threat is posed by a malicious app that impersonates a legitimate app by exploiting app-virtualization as an alternative to repackaging attack. Recalling what this means, exploiting app-virtualization, it is enough to run the legitimate app as a plugin and inserting the malicious code in the host or in a second plugin. Thus, avoiding the burden of analyze the app and bypassing all its countermeasures. In this way, a phishing attack can be launched by intercepting user's input or, more simply, the target app can become a carrier for spreading the malicious code.

Deceived by the appearance of the malicious app, the victim downloads it from the store and then installs it without any additional privileges. We can assume that this malicious app has somehow managed to bypass store examinations, possibly through the use of virtualization.

By doing so, the attacker is able to damage the victim remotely, i.e. without having access to the device.

4.3 Solution Requirements

We wanted a solution that could detect the malicious use of virtualization as an alternative to repackaging. We chose to focus on this attack scenario because we think it is the one that benefits the most from the exploitation of virtualization. In other words, compared to the other attack scenarios, the alternative that does not exploit app-virtualization is far less effective.

In particular, the proposed solution must fulfill the following requirements:

(i) It must be able to detect malicious use of the app-virtualization as an alternative repackaging.

(ii) It must be able to detect whether an application leverages app-virtualization.

(iii) It must be able to work online, i.e. on the user's device, without requiring any special privileges.

(iv) The proposed solution must be able to detect any virtualization framework and not be highly specialized for VirtualApp and DroidPlugin.

(v) It must overcome VAHunt's vulnerability to the use of obfuscation and native code.

5

Design

One of the requirement for our solution is that it must be able to run on the device (requirement (iii)), which means we cannot rely on static analysis. In fact, on an android device it is difficult to inspect the source code (smali code) of other applications. For this reason, we opted for a dynamic analysis by leveraging app-virtualization itself. This solution also allows us to overcome all the limitations of VAHunt related to static analysis, which is the sensitivity to the use of native code and obfuscation techniques (requirement (v)). Usually, dynamic analysis has a big limitation, i.e. the code coverage. In fact, in order to perform efficient dynamic analysis, it is necessary to ensure complete exploration of the app under analysis. In addition, it has been observed that some malware may change behavior over time, and therefore, a too short analysis may also be limiting. In our case, however, the behavior we are looking for is executed immediately on startup. In fact, a malware that exploits virtualization as an alternative to repackaging, must execute the plugin immediately so it can fool the victim. Thus for our analysis it is enough to run the sample for a few moments and, so, we are not affected by the limitations of dynamic analysis.

Using this approach, we based the detection of malicious use of app-virtualization mainly on 3 signatures (requirement (i)). The first one is obtained by applying an hook on the startActivity method. This method is the API responsible for sending intents. Thus, hooking it,

we are able to retrieve information such as package name, component name, action and categories of all intents send by the application under analysis. Doing so, it is possible to detect those aimed at starting activities that do not belong to packages installed on the device, thus suggesting that they are directed to a plugin in the virtual environment. In order to do so, the PackageManager's queryIntentActivities method is used to check whether an application capable of responding to the intent is present in the device.

The second signature collected by Matrioska concerns processes. When a plugin is executed it is done in a different process than the host. Thus, the appearance of a new process is a necessary, but unfortunately not sufficient, condition for detecting virtualization usage. The app being analysed is running in our virtual environment and therefore shares the same UID as our tool. It is easy to obtain a list of active processes that share the same UID as our tool, as it is enough to call the getRunningAppProcesses method of the ActivityManager class.

Lastly, by scanning the internal memory of the app under analysis, we can detect any downloaded or decrypted apk that could be used as plugin. In fact, the apk of any plugin must be stored somewhere at some point. Since the app being analyzed is running in our virtual environment, its dedicated memory area will be no more than a subdirectory of that of our tool. So it is only necessary to periodically scan the files inside it to find any apk. We have noticed that some malware generates and removes the plugin apk very quickly, to the point that it goes undetected by our tool. To overcome this problem, we have also applied a hook to the delete method of the File class, so that we can also examine all files that are deleted.

These 3 signatures are also paired with an analysis of the manifest and the files contained in the assets directory of the application under analysis. In particular, the manifest is searched for the presence of stub components by looking for a large number of components with a similar name. In addition, this analysis is improved by using the processes assigned to components, i.e., a component cannot be a stub component if it is not assigned to a process different from the main one. The total number of processes used by an application is also checked since virtualization cannot be used without declaring processes other than the main one in the manifest. The assets directory is searched for the presence of apk files. Sometimes, in fact, the plugin apk is saved in clear text within the malware. This analysis does not strictly require the use of virtualization, but its use made it easier for us to build the online implementation of the tool. Moreover, in this way we are also able to analyse apps starting from an apk file, without needing

them to be installed on the device first. Thus, to obtain the information we need, we apply a hook to the attachBaseContext method of the Application class. From here we are able to obtain all the information about the components also contained in the manifest and we can access the assets directory. Moreover, this method is always called when an application is started, so we are sure that we will have a chance to retrieve the information we need. With this methodology it is also possible to check the permissions declared by the app under analysis, since an app implementing virtualization often declares a large amount of permissions this could be indicative but we decided not to use this since it has been found that declaring many permissions is a very common practice. An offline version of the manifest and assets directory analysis was also created, which does not use virtualization but instead leverages aapt. This was useful for us to select good candidates on which we could then carry out the full analysis.

This last analysis helps to understand whether an application implements virtualization (requirement (ii)), but it does not provide any data regarding its malicious use as an alternative to repackaging. Actually, manifest analysis is more effective for other attack scenarios. In fact, these signatures searched in the manifest are due to the fact that the host must be compatible with a large number of different plugins of which it cannot know the details in advance. Instead, in the case where virtualization is used as an alternative to repackaging, the host developer knows exactly which plugin will be executed and then he can create a customized manifest that mirrors that of the plugin.

With the goal of making our detection more general and not specialized on the VirtualApp and DroidPlugin frameworks (requirement (iv)), our approach relies on more general, high-level signatures. These signatures alone are not effective enough, which is why we combine many of them in order to improve detection.

A schematic of Matrioska's functionality is given in Figure 5.1.

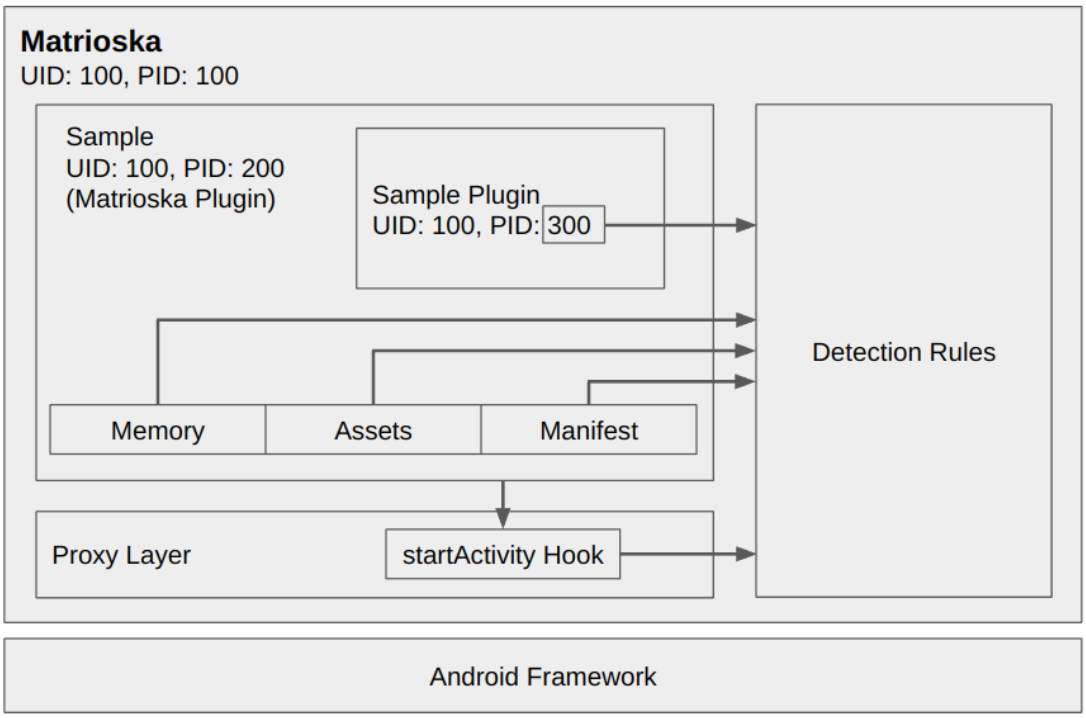

Figure 5.1: Matrioska detection schema.

6

Implementation

Matrioska takes advantage of virtualization to perform a dynamic analysis of the sample. For this purpose, the VirtualApp framework was adopted. We chose the Android 7 version, based on compatibility with the devices at our disposal and the malware in our dataset. Indeed, VirtualApp proved to be very unstable and it was therefore difficult to find a compromise. Furthermore, the hooks were made using the YAHFA [13] framework.

Detect APKs: To determine whether a file is an apk or not, the first 4 bytes of the file itself are first checked, so as to verify that it is a zip. Then, if so, it is checked for the presence of an AndroidManifest.xml file within it. Simply checking the ".apk" extension in the name was indeed limiting.

Detect Stub Components: Information concerning components are used to identify stub components. If a component has a name similar (the longest common substring makes up at least 90% of the name) to at least 5 other components and it has been assigned a different process from the main one, then it is classified as a stub component.

Detection Rules: The result of the malicious detection is expressed by a score ranging from 0 to 3. A score greater than 0 indicates that suspicious behaviour has been detected, the higher the score, the higher the confidence. We have noticed that it is sometimes possible to apply the hook on the Application class, used to collect manifest information, also on the plugin

of the sample. We consider the creation of a second Application class, with package name different from that of the application under analysis, as an evidence of the presence of a plugin. Furthermore, this information is combined with the collected intents to check whether any of them are directed to the package of the second Application class. The score is computed as shown in Algorithm 6.1. Here `spawn_processes` indicates whether at least one additional process was spawned by the analysed app, `spawn_apk` indicates whether apk files were found in the internal memory, `assets_apk` is the number of apks found in the assets directory, `unknown_intent` indicates whether intents directed to packages not installed on the device were detected, `app_classes` indicates the number of Application classes with distinct package names, and `intent+` is true if a match was obtained between one of the unknown intents and one of the additional Application classes, if any.

Algorithm 6.1 Malicious Evaluation

```
score ← 0
if spawn_processes
    if spawn_apk or assets_apk > 0
        score ← score +1
    end if
    if unknown_intent
        score ← score +1
    end if
    if app_class > 1
        score ← score +1
    end if
    score ← score −1
    if intent+
        score ← 3
    end if
end if
return score
```

The result of the manifest and assets analysis is also expressed by a score, this time ranging from 0 to 6. If the score is ≥ 3 there is a good probability that the sample makes use of virtualization, the higher the score, the more this probability increases. Algorithm 6.2 shows the pseudocode of how the score is computed. In the pseudocode, `num_processes` represents the number of distinct processes declared in the manifest, `assets_apk` the number of apks found

in the assets directory, `stub_components` the number of components classified as stubs, and `stub_processes` the number of distinct processes assigned to stub components.

Algorithm 6.2 Score of the Manifest and Assets analysis

```
score ← 0
if num_processes > 0
    if num_processes > 8
        score ← score +1
    end if
    if assets_apk > 0
        score ← score +2
    end if
    if stub_components > 10
        score ← score +1
        if stub_processes > 0
            score ← score +2
        end if
    end if
end if
return score
```

7 Evaluation

We tested Matrioska to assess its accuracy in performing two tasks: identifying the use of virtualization in an app (task 1), and classifying its use as an alternative to repackaging (task 2). In order to do this, 3 datasets was created:

1. The first contains 50 benign popular apps taken from the stores. The purpose of this dataset is to evaluate performance on the first task, in particular, the number of false positives obtained.

2. The second contains 44 benign apps that make use of app-virtualization taken from the store. This dataset also aims to evaluate the first task, but this time the number of false negatives.

3. The third and final dataset contains apps classified as malicious that make use of app-virtualization, and was created for the purpose of evaluating the second task. We started with 18913 malicious applications retrieved from the VirusTotal dataset [14]. These were analyzed with the offline version of the manifest and asset analysis, so as to identify which ones make use of app-virtualization, 277 were identified. Some of these (also based on compatibility with our version of VirtualApp) were manually inspected to search for the behavior related to the alternative repackaging attack. This behavior was

observed in 14 samples. These plus another 36 apps, taken from the 277 apps that make use of virtualization, were combined together to create the third dataset containing 50 malicious apps that make use of app-virtualization.

For comparison purposes, we evaluated VAHunt the same way.

The evaluation of Matrioska was carried out on a Huawei P9 Lite VNS-L31 device with Android 7. The results are shown in Table 7.1 and Figure 7.1.

Matrioska achieves an excellent result in both tasks. In the first task we have only one false positive, this corresponds to the Mozilla Firefox application that we noticed declare components very similar to what we defined as stub. On the other hand, with VAHunt we get poor performance. In the first task we clearly have an anomaly, in fact all 94 samples are classified as positive. In the second task, some of the samples misclassified as false positives can be attributed to the fact that VAHunt locates only the operation of installing a plugin and not executing it. In fact, we have observed that some applications pre-install one or more plugins immediately at startup and, if the user does not access the features that require them, they may even not run at all. Clearly, this behavior is not consistent with our threat model. In general, we can attribute

Matrioska	TP	TN	FP	FN
task 1	44	49	1	0
task 2	14	36	0	0

VAHunt	TP	TN	FP	FN
task 1	44	0	50	0
task 2	7	25	11	7

Table 7.1: Matrioska and VAHunt performances compared in terms of true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN).

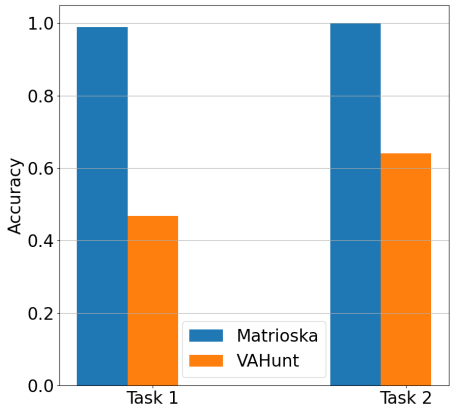

Figure 7.1: Matrioska and VAHunt performances compared in terms of accuracy on our dataset.

the poor performance of VAHunt in the second task to the usage of obfuscation techniques within the samples.

The accuracy value obtained on task number 1 is not the real one since, in a real scenario, the two classes are very unbalanced. In fact, only a small subset of the apps in the stores uses app-virtualization, but in our dataset the two classes contain almost the same number of samples.

7.1 Large Scale App-Virtualization Detection

Then we conducted the app-virtualisation detection analysis (task 1) on large scale. We applied both the Matrioska and VAHunt algorithms on a dataset consisting of 35766 applications, of which 18913 are malware from the VirusTotal dataset [14], and 16853 are from the AndroZoo dataset [15] which contains both malware and non-malware. The results of this analysis are reported in Figure 7.2. The outcome obtained with Matrioska is consistent with what was expected, i.e. that app-virtualization is employed by only a small subset of applications. Actually, the 1.3% is a larger subset than expected, so it is likely to contain false positives. On the other hand, with VAHunt we obtained a surreal result, however consistent with the high number of false positives observed with our accuracy test.

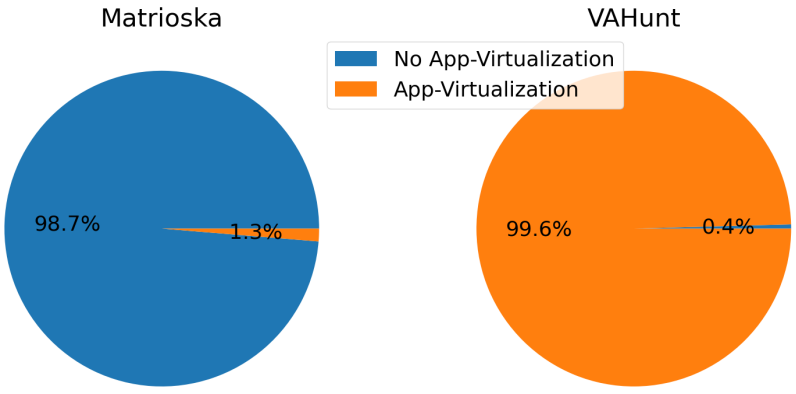

Figure 7.2: Large-scale app-virtualization detection results.

7.2 Cases of Study

In this section, we describe some of the more representative malware that has been analysed. As an example, their behaviour and how they exploit app-virtualization is described in detail, in order to provide a clearer insight into how app-virtualization is exploited for malicious purposes and the challenges we faced.

7.2.1 Case of Study 1 (a.k.a. of3b)

The first malware we are going to look at exhibits the behaviour adopted by all samples that have been classified as malware exploiting app-virtualization as an alternative to repackaging. This malware is a simple host which exploits VirtualApp framework to execute a second apk file. The plugin thus executed is the application that is actually shown to the user, which is why its behaviour has been classified as alternative to repackaging. The plugin apk file is encrypted and stored in the assets directory. Looking at the malware code, we found that the plugin apk is encrypted via AES and that the key is the MD5 hash (actually only a segment of the hash) of the signature of the malware itself. This simple encryption was easy to overcome in a manual analysis but poses a real challenge for an automatic analysis tool.

7.2.2 Case of Study 2 (a.k.a. 1b82)

This sample looks like a mini-game for smartphones and it is classified by third-party anti-malware as a potentially unwanted app (PUA). Its behaviour consists of installing in the virtual environment the 'gamebox.apk' file stored in the assets directory, then it creates a shortcut for the plugin in the home screen. This is repeated whenever the malware does not find the shortcut. This is the annoying/unwanted behavior, as the user is forced to have on the home screen a shortcut for an app he didn't installed. Summarising, this malware exploits virtualization (in particular DroidPlugin) in order to deploy a second app without the user's consent (note that the second app is not installed separately on the device, so if the user uninstalls the malware also the second app is removed). Since we have found other samples exhibiting the same behaviour and installing the same plugin, we believe that the aim of this malware is to promote, or at least spread, this 'gamebox' application.

7.2.3 Case of Study 3 (a.k.a. CleanDoctor)

This malware is an Adware called CleanDoctor, and it appears as a device cleaner. It sets a system alarm in the onCreate() method of the Application class, this alarm is a repeated alarm and it goes off every 3 min. When this alarm goes off, and each time the device completes the boot operation, a specific intent is raised. A receiver captures this intent and triggers the installation in the virtual environment of an advertised apk which is downloaded from a remote server. When the installation is completed, it creates a shortcut on the home screen and executes the plugin. This malware exploits the VirtualApp framework to carry out its malicious behaviour, which corresponds perfectly with that of an Adware.

7.2.4 Case of Study 4 (a.k.a. QihooStore)

This sample is not malware, but we report it as an example of app-virtualization usage that can easily be misclassified as malicious. This application employs a custom app-virtualization framework to install in the virtual environment the apks responsible for providing some features. These apk files are only downloaded and installed if the user chooses to activate the corresponding feature. The only difference between this sample and a malware that exploits app-virtualisation to dynamically load malicious code is the plugin that is loaded.For this reason, in order to detect malware that exploits app-virtualization for the purpose of evasion, it is not enough to analyse how virtualization is used, but instead a full dynamic analysis is required, as for any other malware that implements some dynamic code loading technique.

8 Discussion

8.1 Limitations

Even if the results reported in the previous chapter appear excellent, we believe that there is still much to improve in Matrioska.

First of all, it is mandatory to expand the dataset on which to test our tool. In fact, we were only able to collect and analyze 14 applications that use app-virtualization as an alternative to repackaging. Moreover, these applications turned out to be very similar to each other. This limited dataset is probably the reason why the results look so good.

Also regarding detection we can identify some limitations. For example, our detection of stub components in the manifest can be easily fooled. In fact, an attacker can assign randomly generated strings to components instead of using similar names. Alternatively, in the case of app-virtualization exploited as an alternative to repackaging, the attacker can copy the components names of the application he wants to impersonate. As another example, we noticed that many apps at startup show a "get started" type screen, a malware could hide its behavior behind such a screen fooling both VAHunt and our tool, since it requires user interaction to start the plugin.

8.2 Future Work

For future work, it is necessary to expand and improve the test dataset. In addition, we are planning to add new signatures to be used for detection. For example, we could implement a certificate check of the apks spotted during the analysis. In fact, plugin and host certificates will be issued by different authorities in case the host exploits app-virtualization as an alternative to repackaging. This analysis has already been proposed, but it would be more powerful included in matrioska since it can also retrieve apks that are decrypted or downloaded at runtime.

9 Conclusion

In conclusion, we developed Matrioska, a tool that can detect the malicious use of virtualization as an alternative to repackaging, that is, when it is used to deceive the user by hiding malicious behavior behind a legitimate application. Our tool needs further testing and tuning, but we have already demonstrated the effectiveness of our approach. In addition, we have proven that our approach can also work online, i.e., on the user's device.

References

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

[2] 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*, 2019, pp. 222–235.

[3] L. Zhang, Z. Yang, Y. He, M. Li, S. Yang, M. Yang, Y. Zhang, and Z. Qian, "App in the middle: Demystify application virtualization in android and its security threats," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 3, no. 1, pp. 1–24, 2019.

[4] 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*, 2018, pp. 61–64.

[5] 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*, 2020, pp. 25–32.

[6] 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*, 2020, pp. 535–549.

[7] A. Bianchi, Y. Fratantonio, C. Kruegel, and G. Vigna, "Njas: Sandboxing unmodified applications in non-rooted devices running stock android," in *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2015, pp. 27–38.

[8] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. von Styp-Rekowsky, "Boxify: Full-fledged app sandboxing for stock android," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 691–706.

[9] Z. Cong and L. Tongbo. (2016) Pluginphantom: New android trojan abuses 'droidplugin' framework. [Online]. Available: https://unit42.paloaltonetworks.com/unit42-pluginphantom-new-android-trojan-abuses-droidplugin-framework/

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

[11] 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*, 2018, pp. 50–61.

[12] Google play protect. [Online]. Available: https://developers.google.com/android/play-protect

[13] (2022) Yahfa. [Online]. Available: https://github.com/PAGalaxyLab/YAHFA

[14] Virustotal. [Online]. Available: https://www.virustotal.com

[15] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 468–471. [Online]. Available: http://doi.acm.org/10.1145/2901739.2903508