VAHunt: Warding Off New Repackaged Android Malware in App-Virtualization's Clothing

Luman Shi[†]
Wuhan University
snowmanlu@whu.edu.cn

Jiang Ming[*]
University of Texas at Arlington
jiang.ming@uta.edu

Jianming Fu[*†]
Wuhan University
jmfu@whu.edu.cn

Guojun Peng[†]
Wuhan University
guojpeng@whu.edu.cn

Dongpeng Xu
University of New Hampshire
dongpeng.xu@unh.edu

Kun Gao
Wuhan Antiy Information Technology
gaokun@antiy.cn

Xuanchen Pan
Wuhan Antiy Information Technology
tompan@antiy.cn

ABSTRACT

Repackaging popular benign apps with malicious payload used to be the most common way to spread Android malware. Nevertheless, since 2016, we have observed an alarming new trend to Android ecosystem: a growing number of Android malware samples abuse recent app-virtualization innovation as a new distribution channel. App-virtualization enables a user to run multiple copies of the same app on a single device, and tens of millions of users are enjoying this convenience. However, cybercriminals repackage various malicious APK files as plugins into an app-virtualization platform, which is flexible to launch arbitrary plugins without the hassle of installation. This new style of repackaging gains the ability to bypass anti-malware scanners by hiding the grafted malicious payload in plugins, and it also defies the basic premise embodied by existing repackaged app detection solutions.

As app-virtualization-based apps are not necessarily malware, in this paper, we aim to make a verdict on them prior to run time. Our in-depth study results in two key observations: 1) the proxy layer between plugin apps and the Android framework is the core of app-virtualization mechanism, and it reveals the feature of finite state transitions; 2) malware typically loads plugins stealthily and hides malicious behaviors. These insights motivate us to develop a two-layer detection approach, called *VAHunt*. First, we design a stateful detection model to identify the existence of an app-virtualization engine in APK files. Second, we perform data flow analysis to extract fingerprinting features to differentiate between malicious and benign loading strategies. Since October 2019, we have tested VAHunt in Antiy AVL Mobile Security, a leading mobile security company, to detect more than 139K app-virtualization-based samples. Compared with the ground truth, VAHunt achieves 0.7% false negatives and zero false positive. Our automated detection frees security analysts from the burden of reverse engineering.

[*]Corresponding authors: jmfu@whu.edu.cn and jiang.ming@uta.edu.
[†](1) School of Cyber Science and Engineering, Wuhan University;
(2) Key Laboratory of Aerospace Information Security and Trust Computing

CCS '20, November 9–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7089-9/20/11…$15.00
https://doi.org/10.1145/3372297.3423341

CCS CONCEPTS

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

ACM Reference Format:
Luman Shi, Jiang Ming, Jianming Fu, Guojun Peng, Dongpeng Xu, Kun Gao, and Xuanchen Pan. 2020. VAHunt: Warding Off New Repackaged Android Malware in App-Virtualization's Clothing. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20), November 9–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3372297.3423341

1 INTRODUCTION

To lure mobile users into downloading malicious apps on their devices, malware developers are always seeking to evade security measures and sneak into mobile app marketplaces. A common way to distribute Android malware used to be repackaging legitimate apps [1–3]. Generally, attackers hijack a benign app's logic by decompiling the original APK and adding malicious payloads before distribution; users are fooled into installing the repackaged malware because they believe that they are running the original app. However, since 2016, we have observed a disturbing new trend that malware capitalizes on the latest app-virtualization progress as a new repackaging practice, which causes malware distribution to become easier and stealthier than other traditional approaches.

In an app-virtualization solution, the host app provides a virtual environment on top of the Android framework by creating system service proxies. As a result, the host app can directly launch other

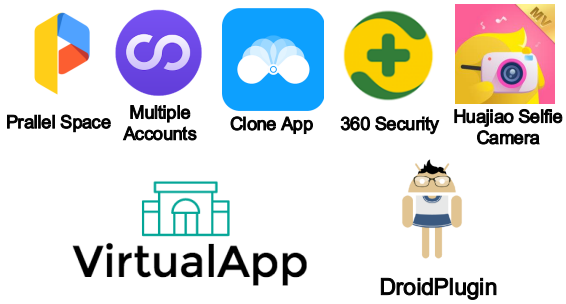


Figure 1: Popular app-virtualization apps and engines.

APK files, called guest apps or plugins, without the trouble of installation. Similar to repackaging, app-virtualization also adds extra code in the form of a plugin, which affects the execution logic of the host app. This new technique has been applied to a variety of applications and gains tens of millions of users [4]. Figure 1 lists some popular app-virtualization-based apps and two open-source virtualization engines (VirtualApp [5] and DroidPlugin [6]). An outstanding advantage of app-virtualization is to allow a single device to run multiple instances of the same app simultaneously [7], such as running two WhatsApp accounts on one phone. Powered by this innovation, the so-called "dual-instance" apps [8] are all the rage in various app markets; the most-downloaded one, Parallel Space [4], has more than 100 million downloads on Google Play. Besides, developers using app-virtualization can also favor modular programming [9], reduce the APK file size [10], and bypass the 65,535 methods limitation of a single DEX bytecode file [11]. For example, the 360 Security app in Figure 1 includes multiple function modules (e.g., memory cleaner, smart performance booster, and anti-virus scanner), and they all exist as plugins to load and use, reducing the coupling of each module.

However, as would be expected, it never takes cybercriminals long to catch on the latest technique trend; they have utilized app-virtualization to repackage and load malicious plugins silently. For example, security experts find that the new version of Trojan *PluginPhantom* has evolved from using traditional repackaging style to app-virtualization [12]: it implements each malicious function as a plugin (e.g., intercept incoming phone calls and upload stolen data) and utilizes an app-virtualization platform to schedule and control its plugins. In this way, PluginPhantom achieves more flexibility to update its modules without reinstalling apps. Besides, all of its malicious payload plugins are further encrypted to frustrate static analysis. As a result, only the executable code of the virtualization framework is statically visible, posing a huge challenge for the current anti-virus engines in the industry. Another example is malware *HummingBad*, which once infected over 10 million Android devices worldwide in 2016. HummingBad's new variant has resurfaced with using DroidPlugin [6] to perform the advertising fraud even more efficiently than its predecessor. 20 Android apps infected by HummingBad were already downloaded by over 12 million times before the Google Security team removed them from the Play Store [13]. Our malware tracing study also confirms that the increment of app-virtualization-based malware has exceeded repackaged malware since the second quarter of 2017, and the gap keeps growing (see Figure 4(b)). The rapid propagation of this new repackaging style indicates that app-virtualization will become the next generation of Android malware distribution channel.

Unfortunately, existing defense techniques are insufficient in the face of new app-virtualization-repackaged malware. Significant efforts remain to be done to understand and defeat this emerging threat. Recent work focuses on the security threats caused by app-virtualization [14–18]. They demystify the underlying mechanism of app-virtualization and highlight the app-virtualization environment is not secure. Due to the share of UID, adversaries can inject malicious code easily and conduct various attacks stealthily, such as privilege escalation, privacy leakage, and phishing. Both *PluginKiller* [14] and *DiPrint* [17] have presented viable solutions to detect app-virtualization environment at run time. However, these dynamic detection heuristics are mainly used to protect benign apps when they are loaded by host app as plugins. The app-virtualization-repackaged malware, like PluginPhantom or HummingBad, is a self-contained system that does not load any third-party plugins. Moreover, all of malicious payload plugins are either encrypted (e.g., PluginPhantom) or downloaded from Internet after the program starts running (e.g., HummingBad). Therefore, static analyses that attempt to inspect malicious payload plugins become futile as well. Zhang et al.'s study [16] demonstrates that app-virtualization-repackaged malware can effectively evade both anti-virus scanners and existing repackaged app detections that rely on measuring code/interface-layout similarities [19–23].

To help security analysts and app market maintainers to defeat malware threats armored by app-virtualization, in this paper, we explore the inner mechanism to make a verdict on app-virtualization-based apps prior to run time. Our in-depth study results in two key observations. **First**, the proxy layer (i.e., the virtualization engine) between plugin apps and the Android framework is the core of app-virtualization mechanism; it creates system service proxies and wraps the plugin components with pre-defined stub components to maintain the plugin lifecycle. Despite the various implementations, the proxy layer transmits data via Intent wrapping/unwrapping in the IPC (Inter-Process Communication) between plugins and system services, and this common behavior reveals the feature of finite state transitions. **Second**, to conceal malicious activities, malware apps typically load and execute plugins stealthily without any user interactions, which is the so-called "self-hiding behavior" [24].

These insights motivate us to develop a two-layer detection approach. **First**, we design a stateful detection model to identify the existence of an app-virtualization engine in APK files. In particular, we abstract the behavior of *wrapping the plugin app's Intent with the predefined stub component* as a finite state machine (FSM) model. Then, we extract all reachable Intent-related FSMs and their associated stub components to check whether they match with the reference pattern. **Second**, we perform data flow analyses to extract stealthy loading and app hiding strategies to differentiate between malware and benign ones. Based on FlowDroid [25], we analyze the data flow and call graph that are related to path APIs and file objects. We summarize four loading strategy features to distinguish malicious app-virtualization apps from benign ones. If an app reveals both an app-virtualization engine and stealthy loading features, we take it as a sample of app-virtualization malware.

The novelty of our design is to detect malicious app-virtualization-based apps from a new standpoint: the way that the virtualization engine loads plugins is a prominent malware feature, even if the plugin's APK file is encrypted or will be downloaded from Internet at run time. We develop our two-layer detection model as an open-source tool, named *VAHunt*. Since October 2019, VAHunt has been deployed into Antiy AVL Mobile Security [26] to evaluate its accuracy of malware detection. We have tested VAHunt with 139K app-virtualization apps, and VAHunt shows a very high accuracy with 0.7% false negatives and zero false positive. The small false negatives come from several game fraud apps, which have an interface to allow game fraudsters to modify game properties. In a nutshell, we make the following three major contributions:

- Although repackaged Android malware via app-virtualization has been on the rise for a while now, limited research work studies the specific countermeasures. We hope our in-depth study paints a cautionary tale for the security community on this imminent threat.
- We zoom in on the inner mechanisms of this new threat and unveil a two-layer detection model, which detects the presence of app-virtualization engine and the self-hiding loading strategies of malware. Our work offers a viable solution to avoid app-virtualization repackaged malware sneaking into Android app stores.
- We have evaluated the efficacy of our prototype, VAHunt, in an anti-malware production environment. To inspire other researchers to explore more comprehensive defensive strategies, we release VAHunt's source code at https://github.com/whucs303/VAHunt.

2 BACKGROUND & RELATED WORK

In this section, we first present the background information needed to understand the Android app-virtualization technique. Then, we summarize the existing work on the security threat study of app-virtualization and their limitations. At last, we introduce the work most germane to our detection methodology.

2.1 Technical Basics of App-Virtualization

In computing, virtualization generally refers to techniques that run virtual machines on versatile platforms [27]. Besides the common hypervisor-based virtualization like VMWare [28] and QEMU [29], app-virtualization is used to isolate individual apps from the underlying Android OS and other apps [30–33]. Since 2015, app-virtualization emerges as a new technique that can load arbitrary third party APKs without installation and modification. The host app creates a virtual environment for plugin apps by dynamic proxy, and it relies on API hooking and binder proxy to bypass system service restrictions. The host app hooks API invocations of plugin apps so that the Android system thinks that all API requests and components are from the host app. Meanwhile, the host app predefines stub components and permissions for plugin apps, and it encapsulates plugin components in stub components at run time. In this way, multiple instances of the same app are able to bypass the UID restriction and run simultaneously.

Please note that the dynamic code loading (DCL) [34, 35] sounds similar to the app-virtualization technique because both of them can load extra code. However, the DCL mostly invokes the methods in an APK file and cannot support the component lifecycle, while app-virtualization represents a more advanced technique to load a whole APK file. This paper focuses on the app-virtualization techniques represented by VirtualApp [5] and DroidPlugin [6]. Two related research tools, Boxify [30] and NJAS [31], have been obsolete in Android 6.0 and later versions. Shi et al. [17] have compared them with VirtualApp/DroidPlugin and summarized the limitations of Boxify and NJAS, such as lacking compatibility and robustness. We attribute the rise of app-virtualization-based malware to the power and availability of the new techniques represented by VirtualApp/DroidPlugin.

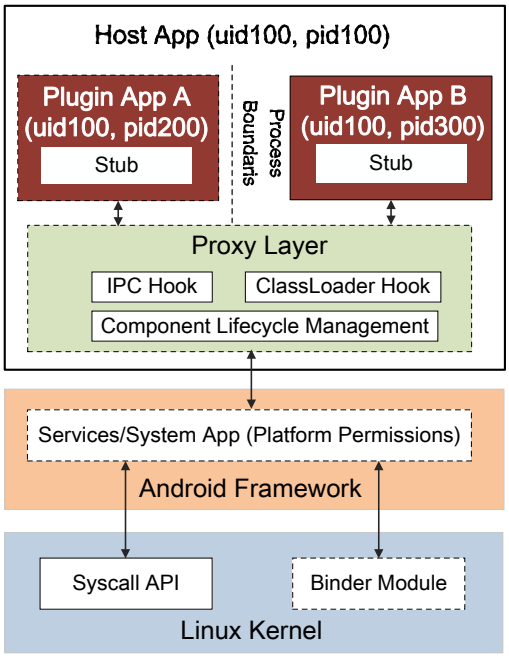


Figure 2: App-virtualization architecture. The host app and plugin apps run in different processes but share with the same unique user id (UID).

VirtualApp and DroidPlugin are the two most popular app-virtualization engines. They both support running multiple copies of the same app on a single device. Although the implementation details are of little difference[1], their key design ideas are quite similar. Figure 2 outlines a typical app-virtualization architecture. The host app provides an independent execution space for each plugin app; it loads multiple plugins with different process IDs, but they share the same UID with the host app. The core of app-virtualization is a proxy layer (a.k.a. virtualization engine), which locates between plugin apps and the Android framework. The proxy layer heavily relies on hooking mechanisms to deceive both Android system services and plugins. For example, it hooks ClassLoader to load plugin's DEX file, and it hooks IPC to maintain the lifecycle of the plugin app's components. In particular, an app-virtualization framework reveals the following common mechanisms.

[1]DroidPlugin uses Service to realize IPC between the host app and plugins, while VirtualApp adopts ContentProvider, which is more suitable and concise for synchronous IPC.

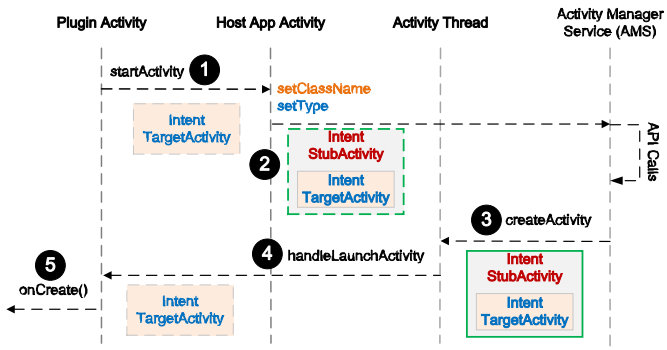


Figure 3: Starting the plugin's Activity by wrapping/unwrapping "TargetActivity" with "StubActivity".

Shared UID. 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. As shown in Figure 2's app-virtualization environment, although the host app and plugin apps are running in different processes (pid 100~300), they share the same UID (uid 100). As a result, different plugins also have the same list of permissions with the host app.

Excessive Permissions. Host apps have to apply for a plethora of permissions to satisfy the plugin's permission demand as much as possible. For example, VirtualApp applies for up to 186 permissions by default, and the average number is 129 in app-virtualization-based apps [17], while a benign app applies for 5 permissions on average [36]. Apparently, excessive permissions violate the principle of least privilege. A plugin is free to use all permissions that the host app applies, even if it declares *zero* permission.

Predefined Stub Components. The plugin app's components are not registered in the host app's manifest file beforehand, because the host app cannot predicate the specific component names of plugin apps. The host app solves this problem by predefining a set of *stub* components in its manifest file. For example, VirtualApp creates 100 stubs for each of the four components by default.

Component Lifecycle Management. The host app must interact with the Android system to maintain the lifecycle of plugin app components. Taking Activity component as an example, we show how the host app manages the component lifecycle by using predefined stub components and hooking Android system service APIs in Figure 3. For example, app-virtualization hooks ActivityManagerProxy by reflection to rewrite other functions like "handleLaunchActivity". Intent is a runtime binding mechanism that can connect two different components and transmit data in IPC processes. When a plugin starts an Activity (❶ in Figure 3), the host app first intercepts "startActivity" method to encapsulate the plugin app's "TargetActivity" intent into the intent of "StubActivity" (❷). "StubActivity" has been predefined in the host app's manifest file. In this way, the host app can deceive Activity Manager Service (AMS) server into creating a new activity for "StubActivity" (❸). Then, the host app recovers the real Intent of "TargetActivity" by hooking "handleLaunchActivity" API of ApplicationThread class and callbacks of ActivityThread class (❹). At last, the plugin app's Activity component will be launched successfully (❺). The processes of handling Service, Content Provider, and Broadcast Receiver for plugin apps are similar. In VAHunt's detection method, we represent the proxy layer's intent encapsulation behavior (as shown in Figure 3) as a finite state machine model.

2.2 Security Threat Study of App-Virtualization

Despite the popularity of app-virtualization-based apps in Android market, researchers have realized the security problems caused by this new technical progress. Recent works have thoroughly studied its security threats [14–18]. Shi et al. [17] shows that the current app-virtualization design introduces serious "shared-everything" threats to users, which makes severe attacks such as permission escalation and privacy leak become tremendously easier. The concurrent work by Zhang et al. [16] conducts a systematic study with 32 popular app-virtualization frameworks. The authors find that the new app-virtualization technique introduces seven common security risks. Moreover, they highlight that malware has abused the app-virtualization as an "alternative and easy-to-use repackaging mechanism".

To inform users that an app is running in an untrusted app-virtualization environment, Plugin-Killer [14] and DiPrint [17] independently develop an Android SDK, which contains a set of dynamic detection features. App developers that wish to avoid having their services hoisted into an app-virtualization environment can embed Plugin-Killer or DiPrint in their own code. However, app-virtualization-based apps are not necessarily malware; these dynamic detection heuristics cannot distinguish malicious apps from benign ones, and their usages are strictly limited when detecting malware that does not load any third-party plugins.

Zhang et al. [16] propose a simple heuristic to detect repackaged malware armored by app-virtualization. They suppose that the host app loads malicious plugins and at least one benign plugin at run time; they observe that the certificate of the host app is different from that of the benign plugin. However, this certificate comparison method is biased by a very small number of malware samples that Zhang et al. analyzed. In our large-scale evaluation, only 0.3% of malware samples comply with Zhang et al.'s observation. Therefore, a comprehensive study on app-virtualization repackaged malware is necessary and of great practical significance.

2.3 App's Stealthy Behavior Detection

Another line of research related to our work is app's stealthy behavior detection [24, 37, 38]. Android malware performs stealthy operations by hiding malicious behaviors to minimize suspicion and prolong its lifetime. AsDroid [37] utilizes the contradiction between the implemented app actions and user's expected behaviors to detect stealthy behaviors. AsDroid relies on API-based detection of six actions, such as starting a phone call, sending SMS, and inserting data into a sensitive database. It then analyzes user-interface components (for identifying user expectations) to detect stealthy app behaviors. Compared with the inconsistent behavior detection of AsDroid, StateDroid [38] finds that a stealthy attack is conducted by a series of actions in a certain order, which can be abstracted as a finite state transition. StateDroid constructs a FSM model via Horn-Clause verification to analyze stealthy attack actions. The recent work proposes the name of "self-hiding behavior" (SHB) [24] to

represent the malicious behavior of concealing app activities, such as removing traces of suspicious actions and hiding the presence of the app. SHB detection approach [24] traces stealthy actions by performing control flow and static taint analyses; this work also inspires VAHunt's detection on malicious-plugin loading strategies.

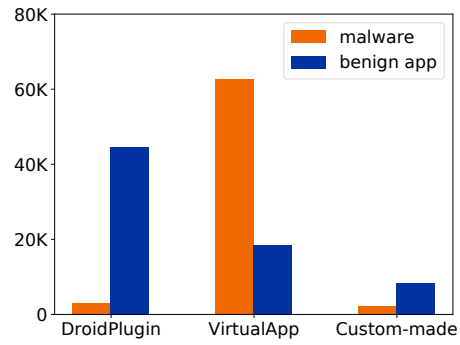
3 NEW REPACKAGED ANDROID MALWARE IN APP-VIRTUALIZATION'S CLOTHING

In this section, we present our in-depth study on tracking the development of app-virtualization-based malware over three years. Our findings unveil the characteristics of this new generation of malware and highlight a pressing need for the security community to design specific countermeasures.

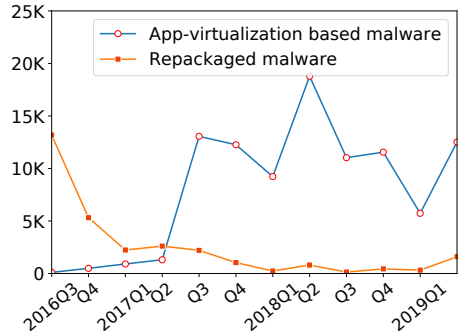
The limitations of the app-virtualization technique, such as lacking permission separation and data isolation, have little impact on malware development. On the contrary, cybercriminals skim the cream off app-virtualization as a new malware repackaging way. As a result, malware development and distribution turn out to be much easier and stealthier than ever before. In particular, app-virtualization repackaged malware reveals the following advantages.

(1) **Bypass anti-virus detection**. App-virtualization enables malicious functions (as different plugins) are decoupled from the host app, and each plugin is either stored under "Assets" directory in an encrypted form or is dynamically downloaded. As long as the host app does not involve any malicious actions, antivirus scanners are difficult to detect it.
(2) **Low development cost**. Traditional repackage still requires quite a heavy reverse engineering effort: adversaries have to modify the original app's code, change the package name using *Apktool* or *DEXlib*, and repackage the app into a new APK file. By contrast, app-virtualization makes repackage much easier: original APK can be directly loaded as a plugin without any modifications.
(3) **Flexible to update modules**. With app-virtualization, malware developers are free to update their modules without reinstalling apps. In addition to updating malicious code, many malware samples begin to add more profitable functions, such as advertisement promotion and pornography propagation.
(4) **Easy to spread**. As mobile developers and users have a huge demand for app-virtualization applications, this fact provides a natural coverage to spread app-virtualization-based malware. For example, they can disguise themselves as a dual-instance app; after a user installs it, the host app dynamically downloads malicious plugins and executes them without the user's awareness.

Since the emergence of app-virtualization repackaged malware in 2016, we are tracking their evolvement all the time. Figure 4(a) shows the total number of app-virtualization-based benign apps and malware samples collected by Antiy AVL in each quarter over three years (2016 - 2019). Antiy AVL's threat detection engine provides rich metadata to identify benign apps, malicious repackaged apps, and app-virtualization-based apps. As DroidPlugin and VirtualApp are two leading app-virtualization engines, we add additional static features to recognize them. For other app-virtualization apps that do not match such fingerprint features, we put them into the "custom-made" category.



(a) The number of app-virtualization-based apps and engines.



(b) The number of new malware samples in each quarter.

Figure 4: The statistics of app-virtualization-based apps.

The two most popular app-virtualization engines, VirtualApp and DroidPlugin, occupy the portion of 92.4% because of their open source. The rest of the apps have custom-made virtualization engines. The core principle of these custom-made engines is similar to VirtualApp/DroidPlugin, but they have different implementation details, such as using AIDL or Content Provider to implement IPC. Custom-made category has 10,621 apps, including six different app-virtualization frameworks. For example, the underlying virtualization engine of Parallel Space is called MultiDroid, and we treat it as a custom-made engine.

Interestingly, the number of malware armored by VirtualApp is about three times larger than that of VirtualApp-based benign apps. In addition, we also investigate the activity level of repackaged malware from September 2016 to December 2019. Figure 4(b) shows the number of new malware samples that have different hash values quarterly. Since 2016, the increment of traditionally repackaged malware slows down, while app-virtualization-based malware exhibits a fast growth from 2017 Q3. Especially, we observe many new malware variants have switched from the previous repackaging style to app-virtualization. The growing gap shown in Figure 4(b) indicates that app-virtualization is very likely to dominate the next generation of Android malware repackaging ways.

We elaborate some detailed characteristics of app-virtualization repackaged malware and present Triada as typical app-virtualization-based malware in Appendix.

4 VAHUNT OVERVIEW

Our in-depth study presented in §2 and §3 results in two key observations. First, the proxy layer between plugin apps and the Android framework is the core of app-virtualization mechanism, and the proxy layer's intent encapsulation behavior (as shown in Figure 3) reveals the feature of finite state transitions. Second, to conceal malicious activities, malware apps typically load plugins stealthily without any user interactions. These insights motivate us to capture a prominent malware feature: the way that the virtualization engine loads plugins. This detection feature is effective even if the plugin's APK file is not statically visible.

We develop a two-layer detection approach, called *VAHunt*. First, we design a stateful detection model to identify the existence of an app-virtualization engine (i.e., proxy layer) in an APK file. Second, we perform data flow analysis to extract self-hiding loading strategies to differentiate between malicious and benign app-virtualization-based apps. Figure 5 presents the workflow of VAHunt. After extracting the necessary information from bytecode and manifest, VAHunt identifies the intent wrapping behavior and searches stub components to detect an app-virtualization engine. After that, VAHunt traces file objects and path APIs to find the features of the stealthy plugin installation. We discuss the details of VAHunt in the following two sections.

5 APP-VIRTUALIZATION ENGINE DETECTION

Activity Manager Service (AMS) manages component schedules in each application. If an app starts an Activity or a Service, it first reports to AMS, and AMS will decide whether to start the Activity or the Service. Since the app and AMS are running in different processes, their Inter-Process Communication relies on Intent [39]. A technical challenge of app-virtualization is to maintain the lifecycle of plugin app components, as all plugins are not installed on the Android system but instead running in a virtual environment. Figure 3 shows how the proxy layer overcomes this obstacle: it wraps the plugin intent with the host app's predefined stub component to deceive AMS into creating a new activity; then it hooks Android system service APIs to unwrap the plugin intent from the stub component and launches the plugin's component finally.

In spite of different kinds of app-virtualization engines are currently in use, such as VirtualApp, DroidPlugin, and custom engines, wrapping the plugin app's Intent with the predefined stub component, as shown in Figure 3, is universal [18]. VAHunt detects this inner mechanism as the presence of an app-virtualization engine.

5.1 Preprocessing

Given an APK file, we first extract the necessary information for our follow-on analyses. We use the tool AAPT [40] to extract Manifest and obtain the component information from an APK file. We get Smali code from a DEX file by using the tool dexdump [41]. Since the number of methods in a DEX file cannot exceed 65536 [11], some apps are split into multiple DEX files. So we merge the Smali code of all DEX files into one file for the convenience of follow-on analyses. Besides, we use FlowDroid [25] to build control flow graphs for each APK file.

5.2 Intent State Machine

We view the plugin intent wrapping behavior, which involves the data transmission via intents, as a finite state machine (FSM) model. The FSM states represent the statuses of intents, and the state transitions are caused by related APIs (e.g., "setClassName" and "setType"). We check whether the intent state machines extracted from each app match with the reference FSM pattern.

```
1 private Intent startActivityProcess(Intent targetIntent)
2 {
3     intent = new Intent(targetIntent);
4     Intent stubIntent = new Intent();
5     stubIntent.setClassName(PackageName, StubActivity);
6     ComponentName component = intent.getComponent();
7     stubIntent.setType(component);
8     startActivity(stubIntent);
9 }
```

Listing 1: JAVA code of wrapping the plugin's Intent.

```
1 <activity
2     android:name="com.lody.virtual.StubActivity$C0"
3     android:configChanges="mcc|locale|touchscreen|..."
4     android:process=":p0"
5     android:taskAffinity="com.lody.virtual.vt"
6     android:theme="@style/VATheme"
7 />
```

Listing 2: The predefined stub Activity in the manifest file.

5.2.1 Intent Operations Extraction. To generate Intent state machines, we need to extract all intent objects and operations. We analyze the Smali code of Android apps to locate the objects with the "android/content/Intent" type and extract all APIs operated on these intents. Specifically, the operations of intent include the creation, attribute setting, and transmission. After an intent is created by "new-instance", we can find operations on the intent object, such as adding flags or setting actions (e.g., "addFlags" and "addAction"). We also record components and other intents related with the intent object. For instance, "setComponent" operation explicitly sets a component name to handle an intent. If the component of an intent is set to a component name coming from another intent, these two intents have a correlation relationship.

5.2.2 Wrapping the Plugin's Intent. As shown in Figure 3, the host app transmits the modified intent to AMS by wrapping the TargetActivity intent in the predefined StubActivity intent. When AMS creates the Activity, the host app recovers the real intent for the plugin. Listing 1 shows the core code of wrapping an Intent. The intent in Line 3 is the targetIntent from the parameter of startActivityProcess() by creating a new instance. The stubIntent is newly created at Line 4, and it has a new Class of StubActivity at Line 5. Line 6 & 7 achieve the goal of wrapping the targetIntent: the targetIntent's component is added to the type of the stubIntent. After that, the encapsulated stubIntent is passed to AMS by "startActivity". In this way, AMS thinks this request of starting an Activity is from the host app instead.

5.2.3 Intent State Transition. By default, an intent is created with an initial state. Its state changes when adding attributes, such as "setClass" and "addFlags". For the transitions related to wrapping an Intent, states of the targetIntent and stubIntent transform with their instances, class names, component names, and type attributes changing. The steps to generate an intent finite state machine (FSM)

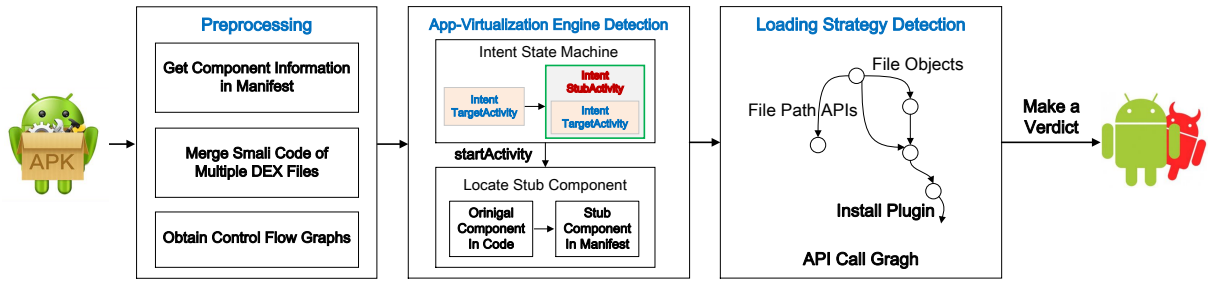


Figure 5: VAHunt system overview. VAHunt makes a verdict on app-virtualization-based apps prior to run time. The two key components are app-virtualization engine detection and plugin loading strategy detection.

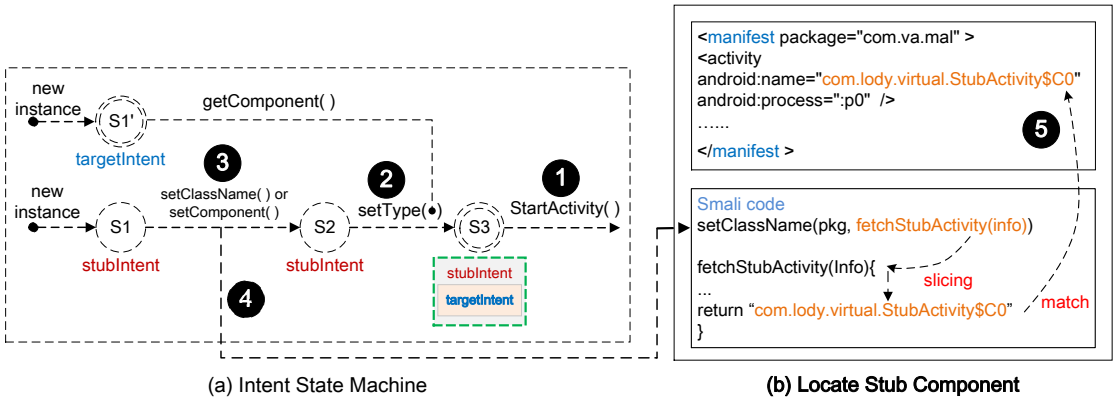


Figure 6: The steps of detecting the app-virtualization engine: 1) match the Intent state machine; and 2) locate stub components.

are as follows. If a statement creates a new intent, the entry of FSM is the new intent object that is represented as a register in Smali code. The state of the new intent object is initialized to the initial state of its corresponding intent state machine. If a statement modifies the attribute of the intent by invoking related APIs, the intent state changes. Otherwise, reading the attributes of intent does not change its state. Besides, VAHunt records related information, such as its class name, function name, and intent operations. When the intent object is sent to AMS by invoking "startActivity" or "startService", the intent state machine comes to an end.

5.2.4 Reference FSM Pattern Match. After generating state machines for each intent, we check whether they match with the reference FSM pattern that represents app-virtualization. As shown in Figure 6(a), this reference FSM pattern abstracts the behavior of wrapping the plugin's Intent in the following three ways: 1) it starts with two source Intent creations (e.g., targetIntent and stubIntent), and stubIntent's final state starts the Activity (❶ in Figure 6); 2) the type of the stubIntent comes from the targetIntent (❷); 3) the stubIntent sets its Class attribute with a new Class name (❸). Once the reference FSM pattern is found, we will perform backward slicing, starting from the stubIntent's new Class name (❹), to confirm that the stubIntent comes from a stub component, which has been registered in the host app's manifest file (❺). In this way, we report the detection of an app-virtualization engine after we successfully match the Intent State Machine model and locate the predefined stub component.

Note that in the typical state machine of app-virtualization (Figure 6(a)), the order of "setClassName" and "setType" is exchangeable, while the "getComponnet" operation of targetIntent must be ahead of "setType" operation. "startActivity" is the prerequisite condition to reach the final state. As the component information can be stored in different ways, "setType" can be replaced by other similar functions like "setData", "setDataAndType", "putExtra" and "putExtras". Furthermore, "setClassName" has the same functionality with the "setComponent"— both of them can set a component name for an Intent. Therefore, "setComponent" is an alternative to "setClassName" in the state transition ❸.

5.3 Locate Stub Component

§5.2 detects the existence of "wrapping the plugin's Intent" behavior in an APK file. Next, we go one step further to confirm that it is the host's predefined stub component to wrap the plugin's Intent.

5.3.1 Component Slicing. After obtaining the final intent state machine information, we trace back the operations on the stubIntent to find the real component name. Generally, the component name exists as a ComponentName type, a ClassName type, or a String type. Specific APIs can convert types, such as using "flattenToString" to convert the ComponentName type to the String type. Developers can set the component attribute of an intent either in the form of a hard-code String or by calling APIs; "setComponent" and "setClassName" are the two most common APIs used to set the component name or the class name of component.

Hence, we trace the final component name with the following steps. We first locate the parameters of "setComponent" or "setClassName" from the information of intent state machine. If the

parameter value is not directly a hard-code String (or ComponentName) but comes from other functions, we perform data flow analysis to find the real component name. For instance, the parameters of "setClassName" may come from the return value of other functions. Specifically, we do backward slicing along the call graph from the parameter of "setComponent" or "setClassName" to collect all APIs used to construct the component name (❹ in Figure 6). As there exists string initialization and concatenation by StringBuilder or StringBuffer, we calculate and record all component name values generated at each transformation.

5.3.2 Stub Component Match. At last, we match the component names extracted by slicing with that of pre-defined stub components. Because the app-virtualization engine has to get the stub components to assign stub intents, it must manage the current available stub components. The typical stub component names in the manifest file are composed of unified Strings and different numbers. As the Listing 2 shows, the stub Activity name is composed of "com.lody.virtual.StubActivity$C" and the number 0. As the components run in different processes, the number in the "android:process" property also differs. Other properties like configChanges, taskAffinity, and theme are all the same with uniform format. All app-virtualization platforms define their stub component names in such a similar style, which is also confirmed by Zhang et al.'s study [16]. Therefore, if one of the collected component names matches with that in the manifest file, VAHunt comes to a decision that an app-virtualization engine is detected.

6 LOADING STRATEGY DETECTION

Given the identification of app-virtualization-based apps, another problem rears its head. We need to further differentiate between malware and benign apps. Unlike the traditional repackaged malware that can be detected by measuring code/interface-layout similarities [19–23], most app-virtualization-based malware samples encrypt their malicious plugins to deter static analyses. To overcome this challenge, we study how malware host apps load plugins silently and hide them after installation. This is the so-called "self-hiding behavior", which has been taken by researchers as a malicious indicator [24, 37, 38]. We aim to find self-hiding features that are required to run malicious plugins without raising suspicion.

6.1 Stealthy Loading Strategy

Figure 7 shows the plugin loading procedures of app-virtualization. For benign apps, after locating the plugin APK's path, plugins are installed with users' consent and execute with normal user interface. However, most app-virtualization-based malware loads plugins silently without any prompt box or user clicking. Some malware variants even hide plugins immediately after installation. Based on these insights, we summarize four stealthy loading characteristics in Table 1. The first two characteristics describe where to load plugin APKs; the next two characteristics summarize the behavior of loading and executing plugins stealthily. We search these four features by performing data flow analysis on call graphs: we propagate particular objects from the starting points (the second column of Table 1) to end points (the third column of Table 1); we find if objects are processed as expected behaviors or calls are invoked

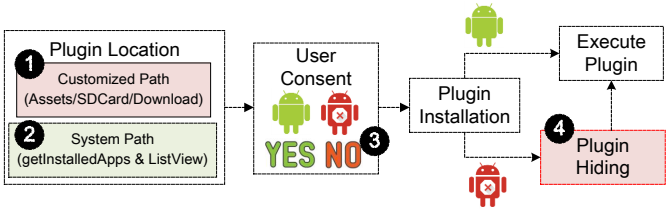


Figure 7: Compared with benign apps, app-virtualization-based malware typically loads plugins without users' consent; they even hide plugins after installation.

with certain parameters. At last, we make decision rules to detect the loading strategy of app-virtualization-based malware.

6.1.1 Load Plugin from Customized Path. The premise of loading plugins in the virtual environment is to know where plugin APKs locate. Most app-virtualization-based malware samples, such as self-contained systems, knows their plugins in advance and loads them directly from a customized location, which includes the public storage that all app can access (e.g., SDCard) and the private data area of the host app, such as Assets subdirectory. Even if plugins are downloaded from the Internet at run time, they are still stored in these locations. The host app gets plugin APK paths either by invoking common Android path APIs or from hard-coded APK paths. The recent work has summarized 22 Android path APIs [42], and we classify them into five categories by different storage locations in Table 2. We analyze call graphs related to these path APIs to search corresponding file objects. After getting the file objects, we further determine whether they are installed by app-virtualization. Generally, app-virtualization provides an installation interface for the host app to install plugins. Developers only need to invoke typical functions (e.g., installApp()/installPackage()) or the overridden startActivity() method of app-virtualization. However, function names of the installation interface can be customized into various names, so we cannot ensure that plugins are loaded by app-virtualization through the fixed strings of the installation interface. Instead, we perform data flow and call graph analysis on file objects until tracing back to Intent wrapping functions (see Figure 6), which demonstrates that the plugin file is installed by app-virtualization.

6.1.2 Load Plugin from System Path. Another category of malware apps that attack benign plugins would lure users into loading benign apps that have been installed on the system. Compared with customized paths, system paths are the isolated storage that host app has no write permission. After requiring a package list, the host app displays user interface elements for users to select which app to load. Then, the host app copies plugin APK from the data space of the original app to its own space and installs the plugin. Generally, developers use getInstalledPackages() of PackageManager to get the installed apps. Hence, we locate getInstalledPackages() in the call graphs as the starting point and track forward to find whether the installed package information is shown in ListView. ListView, often used for the list display, is composed of several items that exhibit words or images. The Android system shows the graphical layout by creating an adapter to show item distributions according to an XML file. When the host app creates a ListView and fills the

Table 1: Four loading strategy characteristics to detect app-virtualization-based malware. Characteristics 1 & 2 describe where to load plugin APKs. Characteristics 3 & 4 describe actions that conceal malware behaviors from being noticed by victims.

Characteristics	Starting Point	End Point
1. Load Plugin from Customized Path	Sensitive Path APIs (see Table 2)	Intent wrapping functions
2. Load Plugin from System Path	getInstalledPackages() Runtime().exec("pm list packages")	onCreateView()/onViewCreated()/ inflate()/onCreateViewHolder()/setContentView()
3. User Consent	installApp()/installPackage()	OnClick()/setOnItemClickListener() Launcher in Manifest ("android.intent.category.LAUNCHER") App lifecycle (onCreate(), onStart(), onReceive())
4. Hide Application	setComponentEnabled(comp, 2, 1) Window.addFlag(FLAG_NOT_TOUCH_MODAL) Window.setFlag(FLAG_NOT_TOUCH_MODAL) Window.requestFeature(FLAG_NOT_TOUCH_MODAL)	Components of the host app

Table 2: Android APIs to obtain the customized file path.

Directory	APIs
/data/data	getDataDirectory(), getCacheDir(), getFilesDir(), getDir(), openFileOutput(), getFileStreamPath(), getDataDir()
/sdcard/	getExternalStorageDirectory(), getObbDir(), getExternalStoragePublicDirectory(), getExternalCacheDir(), getExternalFilesDir()
SQLiteDatabase	openOrCreateDatabase(), openDatabase(), openOrCreateDatabase()
SharedPreferences	getSharedPreferences(), getPreferences(), getDefaultSharedPreferences()
Others	getAssets(), getCanonicalPath(), getPath(), getAbsolutePath()

packages information in the list by setAdapter(), we search the related XML file to find the real items of GUI. If the items include any words or images, that means the behavior of obtaining the plugin packages is visible to users.

6.1.3 User Consent. Typically, benign app-virtualization apps require users to click installation buttons or pop up dialogs to acquire users' consent at the plugin installation time. However, most app-virtualization-based malware samples load their malicious plugins without any user interaction from the very beginning. To detect the non-existence of user clicks, we first locate the plugin installation interface and then trace backward from the interface to the startup of an app. The app startup is usually the component with "android.intent.category.LAUNCHER" label in the manifest file. If we find that the beginning of plugin installation is in the component lifecycles (e.g., onCreate() of Activity), we stop chasing the API call chain. When users agree to load plugins, they often click buttons or items on the screen. So we collect the call chain from the plugin installation and see whether it includes onClick() or onItemClick(). If the call chain reaches the application entrance but showing no user interaction, that means the host app loads the plugin silently. We also deal with thread interruption problem in the call chain analysis in Appendix §B.

6.1.4 Hide Application. After installation, benign apps add their icons to the home screen for users' convenience of the next use. However, malware instances tend to hide themselves and run in the background without showing any icon or Activity. As a result, users cannot see any GUI on the screen. Malware either modifies manifest file to remove the app from the default launcher or invokes setComponentEnabledSetting() to disable icon during running [24]. To hide Activities, malware has two options: 1) malware runs as a Service in the background; 2) malware makes the Activity's main layout transparent and removes the action bar and window title by using the "FLAG_NOT_TOUCH_MODAL" flag. The host app hides its own components to be stealthier in the same way. We perform data flow analysis backwards to investigate the parameters of these special APIs and ensure that the host app configures the components involved in the APIs to hide plugins and itself. For example, as shown in the last row of Table 1, when the three parameters of "setComponentEnabledSetting" are set as startup component of host app, 2, and 1, the outcome of this API is to hide app icons.

6.2 Decision Rules

We take stealthy plugin loading and app hiding actions as malicious behaviors because they violate Google Developer Content Policy [43]. As shown in Figure 7, the malware host app could load plugins from a customized path (❶) or load a benign app to attack it from a system path (❷). Considering an app may contain an app-virtualization engine but never uses it, features ❶ and ❷ are used to confirm that the host does indeed load plugins using the app-virtualization engine. Based on that, most malware host apps install their malicious plugins silently without users' consent (❸), leaving no visible footprints in the user interface. We define a decision rule, *silentInstall*, to detect the malicious loading strategy. *silentInstall*'s formula is: (❶ OR ❷) AND ❸. As long as an app reveals an app-virtualization engine and *silentInstall*, VAHunt labels it as the app-virtualization-based malware.

In addition, we also interested in malware that takes more stealthy actions by hiding icons or Activities after installation (❹ in Figure 7), which we represent as *silentInstall+*, and its formula is: *silentInstall* AND ❹. In our evaluation, we also measure malware exhibiting *silentInstall+* behavior. If an app hides application but does not install plugins stealthily, we report it as a suspicious case.

7 EVALUATION

We have presented our preprocessing to an APK file in §5.1. We use AAPT [40] and dexdump [41] to extract the necessary information for VAHunt. The prototype of VAHunt includes two components: an app-virtualization engine detector and a malicious loading strategy detector. The whole chain of VAHunt has 7,052 lines of python code. All of our experiments are performed on a laptop with one Intel

Core i9-8950HK processor and 32GB memory, running Windows 10 Pro. Our evaluation aims to answer three research questions:

RQ1: How accurately can VAHunt detect app-virtualization-based apps that are built on various virtualization engines (e.g., VirtualApp, DroidPlugin, and custom-made engines)?

RQ2: How accurately can VAHunt further differentiate between malicious and benign app-virtualization-based apps?

RQ3: How well does VAHunt perform in comparison with existing dynamic and static solutions?

7.1 Datasets

Our test cases include three datasets to evaluate VAHunt's detection accuracy. To the best of our knowledge, our evaluation is the largest evaluation to test app-virtualization-based apps so far.

Officially Labeled App-Virtualization-Based Samples. Since October 2019, we have deployed VAHunt into Antiy AVL Mobile Security [26], a leading mobile security company, to evaluate its detection accuracy. At the time of writing, we have tested 139, 358 app-virtualization-based samples that are collected by Antiy AVL over three years by dynamic/static approaches and manual reverse engineering. Most benign apps in the ground-truth dataset are from major APP markets, such as Google Play, 1Mobile Market, Tencent App Gem, and Qihoo 360 Mobile Assistant. Many app-virtualization-based malware samples have been removed from Google Play [44–46]. At the very beginning, the app-virtualization-based malware samples were reported by victims because they bypassed malware detection engines. Security analysts manually reverse-engineer these samples to extract both dynamic and static detection features. Given a suspicious sample, AVL's threat detection engine first uses a machine learning model to match similar syntactic feature vectors. If it does not have similar static features, the AVL engine will run each sample for 5 to 10 minutes to detect malicious or suspicious behaviors, such as plugin process activities and the network traffic. However, 44.4% of app-virtualization-based apps have no local plugins, and 51.8% of plugins have trigger conditions. For unknown malware that does not reveal suspicious behaviors at runtime, security analysts have to manually reverse-engineer the sample. It typically takes an experienced security analyst 25 to 50 minutes to dissect an app-virtualization-based app. The AVL dataset is accurate but at a high cost.

Taking these 139, 358 labeled samples as a ground-truth dataset, VAHunt focuses on the inner mechanisms of app-virtualization and offers a viable detection prior to run time. According to the underlying virtualization engine, we divide these samples into three categories, VirtualApp [5], DroidPlugin [6], and custom-made engine. We detect both benign and malicious apps in each category, and their specific numbers are shown in Row 2~4 of Table 3. Since VirtualApp and DroidPlugin are open-sourced on GitHub, it is convenient for developers and attackers to reuse their source code with a low cost; some of them even modify the source code to generate custom-made engines. Please note that among the total of 81, 225 VirtualApp-based apps, up to 77.2% of them are malware samples, which indicates that VirtualApp has a better acceptance by malware authors. Another interesting observation is that 4, 320 samples are packed in total, but 4, 276 of them are from benign apps. We attribute the low packing ratio of malware to the fact that the app-virtualization mechanism has already provided a layer of effective protection. For the packed samples, we apply Antiy AVL's commercial unpacking tool to obtain their original DEX files.

Table 3: Accuracy evaluation results of VAHunt's two-layer detection model. The last column is the only previous work to detect app-virtualization repackaged malware. Column 4~6 show the detection accuracy data.

Virtu. Engine	Category	Number	VAHunt1[1]	VAHunt2[2]	Certificate [16]
VirtualApp	Benign	18, 508	100%	100%	11.7%
	Malware	62, 717		99.3%	0.3%
DroidPlugin	Benign	44, 498	100%	100%	17.1%
	Malware	3, 014		98.7%	0.5%
Custom-made	Benign	8, 261	100%	100%	13.7%
	Malware	2, 360		99.2%	0.3%
Dual-Instance	Benign	147	100%	100%	0
No app-virtu.	Benign	1, 638	100%	-[3]	-[3]
	Malware	3, 212			
Overall	Benign	73, 052	100%	100%	14.9%
	Malware	71, 303		99.3%	0.3%
Avg. Time				6.9s	12.1s	0.6s

[1]VAHunt1 is the App-virtualization Engine Detector of VAHunt.
[2]VAHunt2 is the Loading Strategy Detector of VAHunt.
[3]VAHunt2 and [16] do not detect the apps without using app-virtualization.

Dual-instance Apps. Since dual-instance apps allow users to load arbitrary APKs, they are ideal cases to evaluate app-virtualization environment detection tools, such as Plugin-Killer [14] and DiPrint [17]. These tools only take effect when they are embedded into the plugin's code. We search related keywords, such as "dual instance" and "multiple accounts", from Google Play and download 147 dual-instance apps.

Apps without App-Virtualization. To test whether VAHunt causes false positives when detecting apps without app-virtualization, we also download 1638 benign apps from Google Play and collect 3212 malicious apps in April 2020. Moreover, all of these non-virtualization apps are manually verified by AVL security experts to confirm that they do not contain app-virtualization engines. Since this process is very costly, we only obtain 4850 non-virtualization apps. Even so, this number is already much larger than the false-positive tested apps in [16], which only evaluated 180 apps.

7.2 App-Virtualization Engine Detection

Table 3 shows the overall results of our experiments. We classify the samples by app-virtualization engine types and maliciousness. We calculate the detection accuracy in the following formula [47]:

$$Accuracy = \frac{TP+TN}{TN+TP+FN+FP}$$

TP means true positives, and TN represents true negatives. Column 4~6 of Table 3 show the detection accuracy data. The data in Column 4 indicate that VAHunt's results on app-virtualization engine detection are perfect in all cases: no false positives and no false negatives. In this step, VAHunt extracts all operations on Intents to generate Intent state machines, and therefore VAHunt's performance here hinges on the number of available Intents. The running time imposed by App-virtualization Engine Detection is about 6.9s (preprocessing 1.2s, Intent state machine 3.3s, and locate stub component 2.4s) on average.

To get a sense of the complexity of building Intent state machines from an APK file, we investigate the Intent distribution in

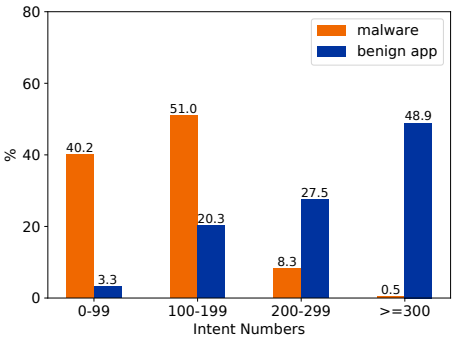


Figure 8: Benign app-virtualization-based apps create more Intents than malware.

app-virtualization-based benign apps and malware. Figure 8 shows that benign app-virtualization-based apps have more Intents than malware (average number: 331 vs. 113). The reason for such a difference is that benign apps typically have more functionalities and embed more third-party packages than malware.

The reference FSM pattern we created involves two correlated intents (see Figure 6(a)), in which the field information of one intent comes from another one. Additionally, we also investigate the occurrence of such multiple-intent related cases inside and outside app-virtualization code. In addition to the intent encapsulation like Figure 6(a), only 4% of our test cases have multiple-intent related cases for the following two scenarios: intent deep copy and shortcut creation/deletion. A deep copy is a fully independent copy of an object, and making changes in the copied object will not affect the original object [48]. Line 3 of Listing 1 shows an example of deep copy by creating a new instance of intent with an existing intent. The "intent" copies all attributes (e.g., Class name and Actions) from "targetIntent", but the two intents are independent. For a shortcut creation, Listing 4 in Appendix shows that the "shortcut" intent informs the Android system to create a shortcut for an app by sending the jump destination in the "intent". After receiving the "shortcut" intent, the Android system creates a shortcut on the screen. When users click the shortcut icon, the app is launched with the information of "intent" that comes from the extra attribute of "shortcut" intent. The difference between multiple-intent related cases inside and outside app-virtualization code is that they transmit different data: only the intent encapsulation behavior transmits the component information of plugins, and VAHunt's locating stub component can rule out other similar multiple-intent related cases. Therefore, our selection of the reference FSM pattern reduces the detection overhead of VAHunt.

7.3 Loading Strategy Detection

Once a sample is detected having an app-virtualization engine, VAHunt moves forward to detect its loading strategy. Column 5 of Table 3 shows the detection accuracy for VAHunt's second step with 12.1s running time on average. For all benign app-virtualization-based apps, VAHunt does not introduce any false positives and false negatives. Note that for the benign apps that use app-virtualization for hot patch, they all notify users in advance, such as prompting "downloading patch file." We find that 1,021 benign apps update

Table 4: The percentage of loading behavior characteristics.

Virtualization Engine	Category	customizedPath	systemPath	silentInstall	silentInstall+[1]
Dual-Instance	Benign	0	100%	0	0
VirtualApp	Benign	93.5%	6.5%	0	0
	Malware	99.6%	0.4%	99.3%	7.6%
DroidPlugin	Benign	90.2%	9.8%	0	0
	Malware	99.9%	0.1%	98.7%	11.7%
Custom-made	Benign	86.3%	13.7%	0	0
	Malware	99.7%	0.3%	99.2%	9.8%

[1]silentInstall+ means host app loads plugins silently and hides app.

Table 5: Most app-virtualization-based apps either do not store plugins locally or encrypt plugins.

Category	No Plugin	Encrypted Plugin	Plain-text Plugin[1]
benign	76.8%	3.7%	19.5%
malicious	10.5%	89.2%	0.3%

[1]Plain-text plugins are APK files that are not encrypted.

the installed plugins, but all of them show user interactions. However, for app-virtualization-based malware, VAHunt introduces about 0.7% false negatives. Upon further investigation, all of these false negatives come from game fraud apps. Game fraud apps load both game modifiers and game apps in an app-virtualization environment, and they have an interface to allow game fraudsters to customize and modify different game properties for quick upgrades. Therefore, game fraud apps exhibit user interactions and violate the stealthy loading strategy. As the customers of game fraud apps are quite limited, we argue that missing the detection of game fraud apps is not a hard limit for VAHunt.

Next, we zoom in on the loading strategy characteristics and decision rules to distinguish malicious apps from benign ones. Table 4 shows the percentage of different loading behaviors in our test cases. Most host apps load plugins from customized paths, but benign app-virtualization apps use more system paths than malware. Especially for the popular dual-instance apps, all of them load third-party apps (e.g., social media apps and game apps) that have been installed on the system. The "silentInstall" column demonstrates that no any benign apps would install their plugins silently. Please note that the "silentInstall+" column indicates that about 10% of malware variants take more aggressive actions to conceal plugins and themselves: in addition to the plugin silent installation, they also hide app icons and Activities, posing a huge challenge for users to perceive abnormals. Table 5 shows the plugin status in app-virtualization-based apps. Apparently, how to store plugins stands in stark contrast between app-virtualization-based malware and benign apps. Most benign apps choose to download plugins dynamically to reduce the APK file size. On the contrary, the majority of malware apps store encrypted plugins in Assets directory for the convenience of distribution.

7.4 Comparison with Existing Solutions

We compare VAHunt with SafetyNet Attestation API [49], PluginKiller [14], DiPrint [17], and Zhang et al.'s work [16]. The first three tools are used to detect the app-virtualization environment

dynamically; as they require developers to add their detection methods in the app code, we use 147 dual-instance apps to load three home-made plugins that embed these three tools, respectively.

SafetyNet Attestation API [49]. Google SafetyNet is an advanced anti-abuse API. They help developers to determine whether their apps are running on a genuine Android device. SafetyNet's "basicIntegrity" testing can identify the signs of a rooted device, emulator, and API hooking. Our result shows that SafetyNet's "basicIntegrity" can detect all of the 147 dual-instance apps' environments dynamically, but it is not fit for detecting app-virtualization-based malware that does not load any third-party plugins.

Plugin-Killer [14] & DiPrint [17]. Plugin-Killer and DiPrint are specifically developed to detect the app-virtualization environment with a set of heuristics, such as undeclared permissions and multiple processes with the same UID. These two tools also successfully recognize all of the 147 dual-instance apps' environments, but they are much faster than SafetyNet—SafetyNet takes a few more seconds to download detection code from the server. Plugin-Killer and DiPrint have the same limitation with Google SafetyNet: they cannot distinguish malicious app-virtualization-based apps from benign ones. By contrast, VAHunt bridges this gap.

App in the Middle [16]. At the time of writing, the only related research work to detecting app-virtualization-based malware is Zhang et al.'s work [16]. They detect the different certificates between the host app and the benign plugin. We also test the accuracy of Zhang et al.'s work in our large-scale evaluation. Although their simple detection heuristic is fast (0.6s on average), the last column of Table 3 shows that their detection accuracy is not acceptable. Only 0.3% of malware samples comply with Zhang et al.'s observation: the host app loads malicious plugins and at least one benign plugin, and the certificate of the host app is different from that of the benign plugin. The encrypted plugins in the host app's subdirectory prevent the extraction of the certificate. Furthermore, this certificate comparison method mislabels all tested dual-instance apps as malware. Dual-instance apps load third-party applications (e.g., WhatsApp) to enrich user experience, and the certificate of the host app is of-course different from the plugin's certificate.

Overhead. The running time imposed by VAHunt's two-layer detection is about 6.9s and 12.1s on average. Including the preprocessing time, VAHunt completes the analysis for each APK file within 30 seconds in our evaluation. The performance bottleneck lies in the multi-round data flow analyses to detect the stealthy loading strategy. Considering VAHunt is an automated detection to free security professionals from the burden of reverse engineering efforts, VAHunt's overhead is acceptable.

8 DISCUSSION & FUTURE WORK

Our work tackles the challenge of detecting app-virtualization repackaged malware prior to run time, but we argue that our results show the challenge is not insurmountable. Our large-scale evaluation in the production environment demonstrates that VAHunt is an appealing technique to complement existing malware defenses. However, VAHunt shares the same limitations with the Android static analysis methods [50–52]. Attackers can encrypt strings in the code (e.g., logs and common strings) to complicate static analyses. In VAHunt's design, we need to match the component names extracted from the Intent state machine with that of pre-defined stub components. We argue that component names in the manifest are hard to be encrypted because the Android system needs to correctly recognize them to manage the component's lifecycle; otherwise, it may lead to an app crash. But if the stub component string in code is encrypted, we cannot match it with that in the manifest. If the malware's installation is concealed beneath an onClick() callback, semantic analysis is required to analyze the context of the button [53]. Using reflection or implementing function in native code could also interrupt call graph generation of VAHunt. A packed app is another long-standing challenge to any static detection methods. If the host app is packed, it will impede VAHunt's analysis from the very beginning. Like what we did in our evaluation, the best practice is to use VAHunt and a generic unpacking tool together.

Detecting malware features at run time has a better resistance against code obfuscation. We also explore the direction of detecting app-virtualization repackaged malware dynamically using similar features with VAHunt. First, we can capture the plugin intent wrapping behavior by API hooking. We use Xposed [54] to hook related APIs that are used to wrap the plugin's intent (e.g., loadDex(), setClassName(), getComponent(), and setType()), and we search the values of live program variables to find the reference Intent state machine pattern like Figure 6(a). Second, to detect the stealthy loading strategy at run time, we run the samples in the experiment environment and observe whether it has the *silentInstall* behavior. However, dynamic detection also suffers from the limited path coverage and dynamic analysis environment evasions. The Triada variant shown in Figure 9 is such a counterexample against dynamic detection, because it is a remote C&C server to control Triada's activities. Without meeting the trigger condition, we cannot observe Triada's loading plugin behavior dynamically.

9 CONCLUSION

Recently, cybercriminals abuse app-virtualization as a new repackaging way, and malware distribution turns out to be much easier and stealthier than ever before. In this paper, we focus on the inner mechanism of this new threat and study the unique features caused by app-virtualization-based malware. We develop VAHunt, a two-layer detection method: 1) we first detect app-virtualization engines in APK files with a stateful model; 2) we conduct data flow analysis to further determine stealthy plugin loading strategies. We have deployed VAHunt into a top anti-virus company to test more than 139K app-virtualization-based samples. Our large-scale evaluation shows that VAHunt reveals very small false negatives and zero false positive. VAHunt significantly reduces the workload of security analysts. We hope the open-source VAHunt inspires more countermeasures against this new generation of repackaged Android malware.

ACKNOWLEDGMENTS

We sincerely thank CCS 2020 anonymous reviewers for their insightful and helpful comments. This research was supported in part by the National Natural Science Foundation of China (61972297, U1636107) and the National Science Foundation (NSF) under grant CNS-1850434.

REFERENCES

[1] Li Li, Daoyuan Li, Tegawendë F. Bissyandë, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro. Understanding Android App Piggybacking: A Systematic Study of Malicious Code Grafting. *IEEE Transactions on Information Forensics and Security*, 12(6), June 2017.

[2] Kobra Khanmohammadi, Neda Ebrahimi, Abdelwahab Hamou-Lhadj, and Raphaël Khoury. Empirical Study of Android Repackaged Applications. *Empirical Software Engineering*, 24(6), December 2019.

[3] Li Li, Tegawende F. Bissyande, and Jacques Klein. Rebooting Research on Detecting Repackaged Android Apps: Literature Review and Benchmark. *IEEE Transactions on Software Engineering*, February 2019.

[4] LBE Tech. How Parallel Space helps you run multiple accounts on Android. http://blog.parallelspace-app.com/how-parallel-space-helps-you-run-multiple-accounts-on-android/, July 2016.

[5] asLody. VirtualApp. https://github.com/asLody/VirtualApp, 2019.

[6] Qihoo360. DroidPlugin. https://github.com/DroidPluginTeam/DroidPlugin, 2019.

[7] JohnC. Mobile App Virtualization: Why the Best Architecture (Should) Always Win. https://sierraware.com/blog/?p=75, May 2015.

[8] Dan Price. How to Run Multiple Copies of the Same App on Android. https://www.makeuseof.com/tag/run-multiple-app-copies-android/, December 2019.

[9] Joe Birch. Modularizing Android Applications. https://medium.com/google-developer-experts/modularizing-android-applications-9e2d18f244a0, August 2018.

[10] Jianqiang Bao. *Android App-Hook and Plug-In Technology*. CRC Press, 1st edition, September 2019.

[11] Google. Enable multidex for apps with over 64K methods. https://developer.android.com/studio/build/multidex, 2019.

[12] Cong Zheng and Tongbo Luo. PluginPhantom: New Android Trojan Abuses "DroidPlugin" Framework. https://dwz.cn/tsm8kSF4, 2016.

[13] Tom Spring. Apps Carrying HummingBad Variant Booted From Google Play. https://threatpost.com/hummingbad-variant-booted-from-google-play/123280/, January 2017.

[14] Tongbo Luo, Cong Zheng, Zhi Xu, and Xin Ouyang. Anti-Plugin: Don't Let Your App Play as an Android Plugin. BlackHat Asia, 2017.

[15] Cong Zheng, Wenjun Hu, and Zhi Xu. Android Plugin Becomes a Catastrophe to Android Ecosystem. In *Proceedings of the 1st Workshop on Radical and Experiential Security (RESEC'18)*, 2018.

[16] Lei Zhang, Zhemin Yang, Yuyu He, Mingqi Li, Sen Yang, Min Yang, Yuan Zhang, and Zhiyun Qian. App in the Middle: Demystify Application Virtualization in Android and its Security Threats. In *Proceedings of the 45th International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'19)*, 2019.

[17] Luman Shi, Jianming Fu, Zhengwei Guo, and Jiang Ming. "Jekyll and Hyde" is Risky: Shared-Everything Threat Mitigation in Dual-Instance Apps. In *Proceedings of the 17th ACM International Conference on Mobile Systems, Applications, and Services (Mobisys'19)*, 2019.

[18] Deshun Dai, Ruixuan Li, Junwei Tang, Ali Davanian, and Heng 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 (SACMAT'20)*, 2020.

[19] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces. In *Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy (CODASPY'12)*, 2012.

[20] Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. Fast, Scalable Detection of Piggybacked Mobile Applications. In *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy (CODASPY'13)*, 2013.

[21] Jonathan Crussell, Clint Gibler, and Hao Chen. AnDarwin: Scalable Detection of Semantically Similar Android Applications. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *Proceedings of the 18th European Symposium on Research in Computer Security (ESORICS'13)*, 2013.

[22] Kai Chen, Peng Liu, and Yingjun Zhang. Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*, 2014.

[23] Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu. ViewDroid: Towards Obfuscation-Resilient Mobile Application Repackaging Detection. In *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks (WiSec'14)*, 2014.

[24] Zhiyong Shan, Iulian Neamtiu, and Raina Samuel. Self-Hiding Behavior in Android Apps: Detection and Characterization. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*, 2018.

[25] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*, 2014.

[26] Antiy AVL Mobile Security. Guarding the Security of Mobile Intelligence Era. https://www.avlsec.com/en/home, [online].

[27] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[28] VMware. VMware Workstation. https://www.vmware.com/, [online].

[29] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the 2005 Annual Conference on USENIX Annual Technical Conference (ATC'05)*, 2005.

[30] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. Boxify: Full-fledged App Sandboxing for Stock Android. In *Proceedings of the 24th USENIX Conference on Security Symposium (USENIX Security'15)*, 2015.

[31] Antonio Bianchi, Yanick Fratantonio, Christopher Kruegel, and Giovanni 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 (SPSM'15)*, 2015.

[32] Chaoting Xuan, Gong Chen, and Erich Stuntebeck. DroidPill: Pwn Your Daily-Use Apps. In *Proceedings of the 12nd ACM ASIA Conference on Computer and Communications Security (ASIACCS'17)*, 2017.

[33] Thi Van Anh Pham, Italo Ivan Dacosta Petrocelli, Eleonora Losiouk, John Stephan, Kévin Huguenin, and Jean-Pierre Hubaux. HideMyApp: Hiding the Presence of Sensitive Apps on Android. In *Proceedings of the 28th USENIX Conference on Security Symposium (USENIX Security'19)*, 2019.

[34] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *Proceedings of the 21th Network and Distributed System Security Symposium (NDSS'14)*, 2014.

[35] Zhengyang Qu, Shahid Alam, Yan Chen, Xiaoyong Zhou, Wangjun Hong, and Ryan Riley. DyDroid: Measuring Dynamic Code Loading and Its Security Implications in Android Applications. In *Proceedings of the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'17)*, 2017.

[36] Pew Research Center. An Analysis of Android App Permissions. http://www.pewinternet.org/2015/11/10/an-analysis-of-android-app-permissions/, 2015.

[37] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. AsDroid: Detecting Stealthy Behaviors in Android Applications by User Interface and Program Behavior Contradiction. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*, 2014.

[38] Mohsin Junaid, Jiang Ming, and David Kung. StateDroid: Stateful Detection of Stealthy Attacks in Android Apps via Horn-Clause Verification. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC'18)*, 2018.

[39] Google. Intents and Intent Filters. https://developer.android.com/guide/components/intents-filters, 2019.

[40] Android AAPT. https://androidaapt.com/, 2019.

[41] Android dexdump. http://manpages.ubuntu.com/manpages/xenial/man1/dexdump.1.html, 2019.

[42] Chris Chao-Chun Cheng, Chen Shi, Neil Zhenqiang Gong, and Yong Guan. EviHunter: Identifying Digital Evidence in the Permanent Storage of Android Devices via Static Analysis. In *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*, 2018.

[43] Google. Google Developer Content Policy. https://play.google.com/about/developer-content-policy.html, 2020.

[44] Swati Khandelwal. Nasty Android Malware that Infected Millions Returns to Google Play Store. https://thehackernews.com/2017/01/hummingbad-android-malware.html, 2017.

[45] Rafia Shaikh. Chinese Ad Company That Turned Out to Be a Cyber Crime Group Is Back with "a Whale of a Tale". https://wccftech.com/hummingwhale-android-malware/, 2017.

[46] Cong Zheng, Wenjun Hu, and Zhi Xu. A New Trend in Android Adware: Abusing Android Plugin Frameworks. https://researchcenter.paloaltonetworks.com/2017/03/unit42-new-trend-android-adware-abusing-android-plugin-frameworks/, 2017.

[47] Aswathi B.L. Sensitivity, Specificity, Accuracy and the relationship between them. http://www.lifenscience.com/bioinformatics/sensitivity-specificity-accuracy-and, 2009.

[48] Joe. Java Clone, Shallow Copy and Deep Copy. https://javapapers.com/core-java/java-clone-shallow-copy-and-deep-copy/, 2014.

[49] Google. SafetyNet Attestation API. https://developer.android.com/training/safetynet/attestation, 2019.

[50] Lei Xue, Xiapu Luo, Le Yu, Shuai Wang, and Dinghao Wu. Adaptive Unpacking of Android Apps. In *Proceedings of the 39th International Conference on Software Engineering (ICSE'17)*, 2017.

[51] Yue Duan, Mu Zhang, Abhishek Vasisht Bhaskar, Heng Yin, Xiaorui Pan, Tongxin Li, Xueqiang Wang, and XiaoFeng Wang. Things You May Not Know About Android (Un) Packers: A Systematic Study based on Whole-System Emulation. In *Proceedings of the 25th Network and Distributed System Security Symposium (NDSS'18)*, 2018.

[52] Yueqian Zhang, Xiapu Luo, and Haoyang Yin. DexHunter: Toward Extracting Hidden Code from Packed Android Applications. In *Proceedings of the 20th European Symposium on Research in Computer Security (ESORICS'15)*, 2015.
[53] Shengqu Xi, Shao Yang, Xusheng Xiao, Yuan Yao, Yayuan Xiong, Fengyuan Xu, Haoyu Wang, Peng Gao, Zhuotao Liu, Feng Xu, and Jian Lu. DeepIntent: Deep Icon-Behavior Learning for Detecting Intention-Behavior Discrepancy in Mobile Apps. In *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS'19)*, 2019.
[54] rovo89. Xposed Module Repository. https://repo.xposed.info/, 2019.
[55] Avast Threat Intelligence Team. Malware posing as dual instance app steals users' Twitter credentials. https://blog.avast.com/malware-posing-as-dual-instance-app-steals-users-twitter-credentials, 2016.

APPENDIX

A DETAILED INFORMATION ABOUT OUR STUDY IN SECTION 3.1

A.1 Characteristics

We summarize the common characteristics of app-virtualization repackaged malware. The first two characteristics cover how malware utilizes app-virtualization to perform malicious behaviors; the next two characteristics are about how malware host apps store and load plugins.

Self-contained Systems. In this category, malware authors encapsulate an app-virtualization engine and malicious plugin modules as a self-contained system. These malware instances are spread using social engineering tricks to deceive users into downloading them. Once on the victim's device, malware can load the malicious plugins either from its own subdirectory like PluginPhantom [12] or download plugins from Internet like Hummingbad [44]. These plugins can perform various malicious functions, such as intercepting incoming phone calls and SMS, advertisement/pornography promotion, and ransomware behavior. As this kind of malware does not load any third-party plugins, existing app-virtualization environment detection heuristics [14, 17] do not work here.

Attack Benign Plugins. The second category of malware exploits the vulnerability of app-virtualization to compromise benign apps. They disguise themselves as an attractive app-virtualization application, such as a dual-instance app, to lure users into installing them. Then, a malicious plugin conducts attacks in the background when the benign plugin is running. Due to the "shared-everything" threat [17], even with zero permission, the malicious plugin can still get access to other plugins' sensitive data and inject malicious code dynamically into other running plugins.

Encrypted/Downloaded Plugins. To impede static analyses that attempt to inspect malicious payload plugins, most malicious plugins' APK files are not statically visible. Our study shows that up to 89.2% of app-virtualization-based malware samples encrypt their plugins, and 10.5% of malware samples choose to download plugins from Internet at run time. For malware samples that encrypt their plugins, we also reverse-engineer the cryptographic algorithms they use. 84.5% of them apply the standard AES algorithm, and the left of them adopt a self-defined encryption algorithm, such as performing XOR operations with specific byte streams.

Load Malicious Plugins Stealthily. Plugin loading strategies stand in stark contrast between app-virtualization-based malware and benign apps. In general, a user interface will show up when a benign app is loading plugins. It is the user's turn to select which plugin to load, and then the host app copies and loads the plugin APK file from the specific path that the user decided. Besides, benign apps do not hide their icons from the phone's desktop. On the contrary, malware loads malicious plugins stealthily to hide their malicious behaviors. Their plugin loading process typically does not involve user interactions (e.g., button click or popup window). The malware host app searches for plugin APKs by invoking path APIs, because it has already known where malicious plugins locate. After installation, malware tends to hide themselves by running in the background without showing any icon or Activity.

A.2 Case Study

A new version of malware Triada[2] is developed on top of DroidPlugin to steal the victim's personal information without raising suspicion. Triada disguises itself as Wandoujia app, which is one of China's most prominent Android app stores. As shown in Figure 9, after installation, Triada reminds users that "the program is incomplete and needs to be reinstalled" to lure users into installing the real Wandoujia app, but Triada runs in the background by hiding its icon. If users disagree, Triada cancels the real Wandoujia's installation but stills runs in the background as a memory cleaning service. Triada hides all APK plugins in its "Assets" directory, and each plugin has a dedicated action. One of its plugins communicates with a remote C&C server, which instructs Triada to load a particular plugin to carry out an activity, such as connecting a WiFi signal, recording calls, acquiring realtime location, sending messages, uploading private data, or updating itself. Besides, all of these plugins run in the background without showing any interfaces.

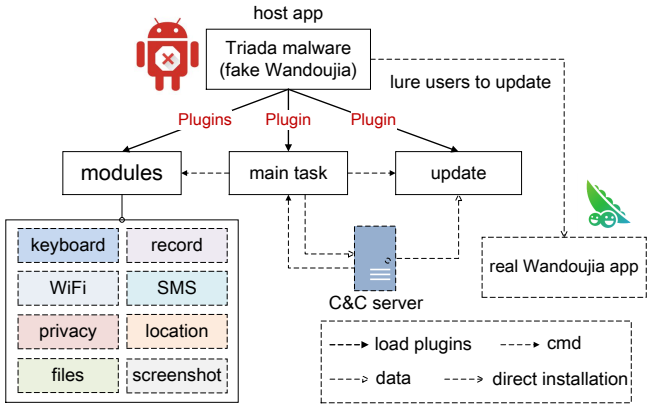


Figure 9: Triada malware disguises itself as Wandoujia app and loads malicious plugins via DroidPlugin.

B THREAD INTERRUPTION ANALYSIS IN SECTION 6

Detecting the user consent characteristic is to analyze the API call chain from plugin installation to app startup. However, our study shows many host apps start new threads to load plugins, which complicates the chasing of API call chain. For example, a VirtualApp-based malware sample steals users' Twitter credentials in a new thread by using Thread Class and then starts the thread in

[2]SHA-1 value: e2b05c8fdf3b82660f7ab378e14b8feab81417f0

another Class [55]. The thread-related invocation is indirect—the actual call destination of Thread.start() is the run() method of the corresponding class, which will interrupt the call chain. Listing 3 shows a typical thread creation using Thread Class. A new thread is started by creating an instance of MyThread (Line 3) and invoking the start() function at Line 4. This causes an indirect invocation to the real task code in the run() method of MyThread Class at Line 8. Besides Thread Class, using Runnable interface and asynchronous tasks (AsyncTask Class) can create threads as well. The implementations of these three thread creations are similar, so we take Thread as an example to explain how we reconnect the interrupted call chain via the indirect invocation solution [37].

```
1 public class Installation {
2     public void goThread() {
3         MyThread thread1 = new MyThread();
4         thread1.start();   //trigger the run () method at Line 8
5     }
6 }
7 public class MyThread extends Thread {
8     public void run() {
9         loadPlugins()  //a time-consuming process
10    }
11 }
```

Listing 3: JAVA code of starting a Thread.

As we generate the call chain backwards, we will first meet the run() method in Thread Class or Runnable interface. After finding the Class name of run(), we traverse the code to find the real caller that invokes the start() method and connect the interruption. AsyncTask Class is an encapsulation of thread based on the thread pool; asynchronous tasks created by AsyncTask Class are asynchronous with UI threads. Once the work in the AsyncTask thread has been done, UI thread will continue to work by invoking onPreExecute(), doInBackground(), onPostExecute() or onProgressUpdate(). We deal with such cases similar with Thread Class to find the real caller.

```
1 public void createShortcut()
2 {
3   Intent intent = new Intent(Intent.ACTION_MAIN);
4   itent.addCategory(Intent.CATEGORY_LAUNCHER);
5   intent.setClass(context, clazz);
6   Intent shortcut = new Intent(Intent.ACTION_CREATE_SHORTCUT);
7   shortcut.putExtra(Intent.EXTRA_SHORTCUT_INTENT, intent);
8   shortcut.setAction("com.android.launcher.action.
      INSTALL_SHORTCUT");
9   context.sendBroadcast(shortcut);
10 }
```

Listing 4: Creating a shortcut involves two related Intents.