"Jekyll and Hyde" is Risky: Shared-Everything Threat Mitigation in Dual-Instance Apps*

Luman Shi‡
Wuhan University
Wuhan, Hubei, China
snowmanlu@whu.edu.cn

Jianming Fu†‡
Wuhan University
Wuhan, Hubei, China
jmfu@whu.edu.cn

Zhengwei Guo‡
Wuhan University
Wuhan, Hubei, China
zhengweiguo@whu.edu.cn

Jiang Ming†
University of Texas at Arlington
Arlington, TX, USA
jiang.ming@uta.edu

ABSTRACT

Recent developed application-level virtualization brings a groundbreaking innovation to Android ecosystem: a host app is able to load and launch arbitrary guest APK files without the hassle of installation. Powered by this technology, the so-called "dual-instance apps" are becoming increasingly popular as they can run dual copies of the same app on a single device (e.g., login Facebook simultaneously with two different accounts). Given the large demand from smartphone users, it is imperative to understand how secure dual-instance apps are. However, little work investigates their potential security risks. Even worse, new Android malware variants have been accused of skimming the cream off application-level virtualization. They abuse legitimate virtualization engines to launch phishing attacks or even thwart static detection.

We first demonstrate that, current dual-instance apps design introduces serious "shared-everything" threats to users, and severe attacks such as permission escalation and privacy leak have become tremendously easier. Unfortunately, we find that most critical apps cannot discriminate between host app and Android system. In addition, traditional fingerprinting features targeting Android sandboxes are futile as well. To inform users that an app is running in an untrusted environment, we study the inherent features of dual-instance app environment and propose six robust fingerprinting features to detect whether an app is being launched by the host app. We test our approach, called *DiPrint*, with a set of dual-instance apps collected from popular app stores, Android systems, and virtualization-based malware. Our evaluation shows that DiPrint is able to accurately identify dual-instance apps with negligible overhead.

CCS CONCEPTS

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

ACM Reference Format:
Luman Shi, Jianming Fu, Zhengwei Guo, and Jiang Ming. 2019. "Jekyll and Hyde" is Risky: Shared-Everything Threat Mitigation in Dual-Instance Apps. In *The 17th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '19), June 17–21, 2019, Seoul, Republic of Korea.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3307334.3326072

1 INTRODUCTION

With the growing trend of Bring Your Own Device (BYOD) [7, 19, 20] and the richness of Android applications, users are paying more attention to personal information privacy. In many scenarios, being able to log in two app accounts simultaneously on the same device is much in demand, e.g., one social networking app account is for personal use, and the other is for business purpose. However, most apps (e.g., Twitter, Linkedin, and Facebook) do not support multiple active instances so that users have to repeatedly login in and out to switch account. Moreover, users can not install two copies of the same apps on a single Android device because of the unique user id (UID) restriction. In general, Android system creates a UID for each app to be installed according to its package name. As a result, many professionals have to carry at least two mobile devices to meet their needs.

The recent Android application-level virtualization developments (e.g., VirtualApp [2] and DroidPlugin [28]) iron out this problem and achieve the goal of "running multiple copies of the same app on Android" [32, 35, 43, 48]. The key idea of Android application-level virtualization is that a host app creates a virtual machine-like environment on top of Android framework, and it relies on Java dynamic proxy mechanism to launch arbitrary apps (called "guest apps") from their APK files without installation. As the host app's virtual environment is transparent to Android system, the actions from a guest app will be treated as the host app actions from Android system's viewpoint. In this way, multiple instances of the

*"Jekyll and Hyde" is a metaphorical term to describe someone with two-sided personalities - one good and one evil. Here we use this term to indicate that the popular dual-instance apps have posed significant security risks.

†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

MobiSys '19, June 17–21, 2019, Seoul, Republic of Korea

ACM ISBN 978-1-4503-6661-8/19/06…$15.00
https://doi.org/10.1145/3307334.3326072

same app are able to bypass the UID restriction and run simultaneously. Note that this technique is fundamentally different from the well known dynamic code loading (DCL) mechanism. Instead of loading a small piece of code (e.g., a dex or jar file) that is tightly coupled to the base app, virtualization technique could load arbitrary APK files that have a complete application entry point and lifecycles. Powered by this innovation, the so-called "dual-instance apps" are getting more and more popular in various Android app markets. Up to now, 42 dual-instance apps are launched in Android app markets and gain high downloads and popularity. Within more than one year from May 2017 to December 2018, downloads of most still-alive dual-instance apps have exponential growth. The most representative one, "LBE Parallel Space", gained as much as 7 million downloads after 3-month global launch in May 2016. Until December 2018, it has been downloaded more than 100 million times from Google Play[1].

In this paper, we argue that the security risks caused by dual-instance apps have been significantly underestimated. The current dual-instance app design indeed meets the high demand of users, but it overlooks the basic Android system security enforcement in permission separation and data isolation. Especially all guest apps share the same UID with the host app. That means these apps also share a common list of permissions, and guest apps may have many additional permissions that they do not declare. Besides, since the strict access control based on different UIDs is missing here, malicious guest apps can acquire the private data of others running in the same virtualization environment without raising suspicion. We call all of these as "shared-everything" threats. New generation malware has capitalized on dual-instance app's design flaws for various malicious purposes. Malicious host apps could steal privacy and update guest apps easily. For example, a new malware sample uses a customized version of VirtualApp as malicious host app to launch phishing attacks and steal users' Twitter credentials by luring users into running Twitter in its control environment [2]; Malware as guest apps loaded by dual-instance apps is quite a lot. PluginPhantom [75] relies on DroidPlugin [28] to install malware from "Assets" directory and steal private files; new adware abuses dual-instance apps to automatically launch different advertisement apps without user interactions [74]. To complicate matters further, recent news has reported that stealthy Android malware can be crafted to evade the anti-virus scanners in VirusTotal[2]. They leverage application-level virtualization to hide malicious behavior in multiple guest apps [4, 75].

Our study shows that many severe attacks (e.g., permission escalation, privacy leak, and component hijacking) become quite straightforward under dual-instance environment, and some attacks may not succeed in real Android system or traditional application-layer sandboxes [5, 9]. However, most Android developers have not yet grasped the seriousness of these potential threats. Some online payment apps are able to detect whether they are running in an Android emulator or root device. For example, Android Pay will disable payment function if it detects itself executing in such sensitive environments [62]. Unfortunately, most critical apps cannot discriminate between dual-instance app environment and Android system, leaving themselves vulnerable to a larger attack surface. The reason is traditional fingerprinting features of Android sandboxes are completely ineffective for dual-instance apps.

To inform users that an Android app is being launched by a host app without installation, we develop a tool named *DiPrint* to automatically fingerprint dual-instance app environment. We study the principle differences between virtualization-based dual-instance apps and Android system. We extract inherent features that are the candidates to indicate dual-instance app environment with static and dynamic analysis. The combination of multiple fingerprinting features requires the design change of virtualization mechanism to be evaded. We evaluate DiPrint with a set of dual-instance apps collected from four popular Android app stores (Google, 1Mobile, Tencent, and Qihoo), various Android systems, and unknown malware collected in the wild. Our results show that DiPrint is able to identify all of the dual-instance app environments without false positives. DiPrint's overhead is negligible as well with an average of 0.36 ms detection time. Furthermore, we demonstrate that dual-instance apps are immune to the traditional detection heuristics of Android sandboxes. DiPrint provides a viable countermeasure for developers that wish to avoid having their services hoisted into an untrusted virtualization environment, and critical Android apps such as mobile payment and banking apps can import DiPrint as a class to escape potential data loss. In summary, we make the following contributions.

- We investigate the "shared-everything" threats caused by dual-instance apps. Our case study shows that many severe attacks can be launched successfully in most dual-instance apps.
- Our research reveals the principle of hot dual-instance apps in depth. The tricks of the underlying virtualization mechanism are not well known.
- We develop a tool called DiPrint to detect dual-instance apps at run time. We study the discrepancies caused by dual-instance app's virtualization mechanism and extract robust fingerprinting features. Other applications can benefit from our lightweight detection code. The source code of DiPrint is available at https://github.com/whucs303/DiPrint.

2 BACKGROUND & RELATED WORK

Android application-level virtualization is an innovative technique that a host app can load and launch any guest app's APK file in a virtual execution environment without installation. With this technique, two copies of the same app can run side-by-side in an unmodified Android system. The virtual machine-like environment created by the host app plays a role of broker to interact with guest apps and Android system services. In one respect, it provides a normal execution environment for guest apps and manages their lifecycle and all requests. On the other hand, it has to conceal the identities of guest apps from system services. To bypass the system's restriction, the broker takes over system services such as AMS (ActivityManagerService) and PMS (PackageManagerService) by intercepting Binder Inter-Process Communication (IPC).

[1] Parallel Space - Multiple accounts & Two face: http://parallel-app.com/
[2] Free online virus, malware, and URL scanner: https://www.virustotal.com

2.1 Previous Work Limitations

Several Android application-level virtualization works have been proposed to intercept API calls and monitor behaviors of guest apps [5, 9, 13, 69]. Boxify [5] leverages Android's "isolated process" feature to build a virtual environment. Process isolation is the segregation of different application processes to prevent others from accessing certain service components and resources. In Boxify, every guest app runs in different isolated processes with much fewer privileges than a regular app process. And the broker is implemented with a reference monitor to mediate over-privileged requests from guest apps to system. To achieve IPC, Boxify leverages Android Interface Definition Language (AIDL) service. However, the asynchronous property of AIDL adds further complications to the synchronous communication between guest apps, leading to low robustness. Different from Boxify, the broker of DroidPill [69] runs in the same process with the guest app. By instrumenting Dalvik Virtual Machine and patching native library's Global Offsets Tables (GOT), the broker takes control of guest apps. NJAS [9] generates a compatible stub application to load code and resources of the original app, and the stub application has the same permissions with the original app. Using the ptrace mechanism, the broker could monitor executions of guest apps through system call interception. However, ptrace has to interrupt the processes of guest app and freeze all threads, which may cause message block and process crash. Furthermore, NJAS can only handle two types of public app objects among the five ones (i.e., Component, Authority, Account Type, Custom Permission, and Intent Action), and it can only launch one guest app. The hooking techniques they adopt make them only work on the lower Android versions from Android 4.1 to 5.1. Boxify's GOT hook is not adapted to the change of API base address calculation since Android 6.0 [30], while both NJAS and DroidPill lack the ability of ART hook. As the mainstream of Android OS has been upgraded to Android 6.0 or the above versions, the lacks of compatibility and robustness greatly limit the adoption of previous application-level virtualization approaches.

In addition, the previous efforts in running two copies of the same app mainly rely on repackaging [6, 17, 18, 31, 50, 68, 73]. It produces a new APK file with the same content but a different package name from the original app. Then the two apps run on a same device with two different UIDs. Repackaging has to locate the original app's important functions and embed extra code in the original app [72]. This violates Android's same-origin model and infringes the third party's property right [51, 77], putting the repackaged versions into a legal gray area. Reference hijacking [71] combines repackaging and rebuilding "framework.jar" to provide a virtualization environment. However, it still has to add additional code to guest apps before loading, which also breaks the integrity of guest apps.

2.2 VirtualApp Innovations

Android application-level virtualization is originally designed for platform openness [37] or flexible installability [40]. The recent developments [2, 16, 28, 38] overcome the drawbacks in the previous work and fully support running multiple instances of the same app in stock Android[3]. The most representative open source platforms, VirtualApp [2] and DroidPlugin [28], have enjoyed a great reputation in Android markets, as many dual-instance apps are built on top of these two virtualization engines. Although VirtualApp, DroidPlugin, or other custom-made tools may have slightly different implementation details, their key design ideas are quite similar. According to our statistics, more than one-third of the dual-instance apps and more than 76% of the dual-instance malware are using VirtualApp. And VirtualApp gains thousands of stars in GitHub. Therefore, our paper takes VirtualApp as an example to introduce how virtualization-based dual-instance apps work. As shown in Figure 1, a dual-instance app (i.e., host app) creates an independent execution space from Android system. When loading guest apps in dual-instance app context, all of them share the same UID with the host app but with different process IDs. By using dynamic code loading and dynamic proxy, host app is able to set up system service proxies to take control of guest app communication with system services and other apps, avoiding the process crash caused by ptrace mechanism. Also, VirtualApp implements the synchronous IPC using the synchronization feature of Content Provider, getting rid of the complicated IPC problem raised by Boxify. VirtualApp's design and implementation are so elegant and stable that it does not rely on particular Android versions, and it can be compatible with the latest Android 9.0 and virtualize most apps in markets.

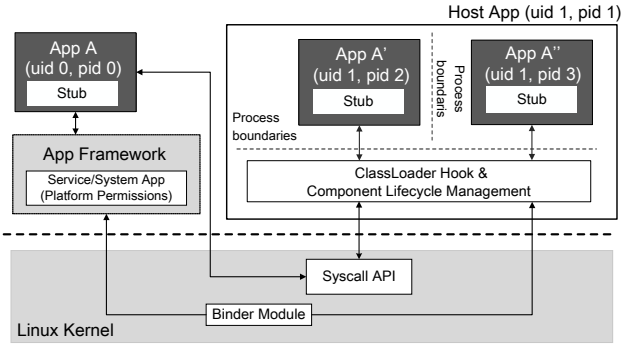

Figure 1: The typical dual-instance app implementation based on VirtualApp. The dual-instance app (i.e., host app) launches two copies of App A as guest apps (App A' and App A") in virtualization environment, while the original App A is running directly in Android system side-by-side. The Host App shares the same UID with App A' and App A".

2.3 Detecting Android Dynamic Analysis Environment

Another line of research related to our work is the detection of Android dynamic analysis environment, including root devices [41, 56] and Android sandboxes [23, 33, 42, 47]. As most sandboxes are system-level emulators, the previous work focuses on finding the discrepancies between Android emulators and real devices [23, 23, 34, 47]. For example, QEMU-based and VirtualBox-based emulators

[3]Stock Android means the vanilla version of Android, which is the most basic version of Android OS designed by Google [53].

exhibit a large number of hardware-related differences with real systems [34, 42]. Rooted devices are identified based on static and dynamic heuristics [33, 41, 56, 59], such as rooting-related files, permissions of certain directories, and the traces of the relevant behavior in the system log. However, the virtualization mechanism of dual-instance app reveals totally different characteristics from existing Android dynamic analysis environment. As a result, the traditional fingerprinting features will fail to detect dual-instance apps.

3 SHARED-EVERYTHING THREATS

The design of virtualization-based dual-instance apps is a "double-edged sword". They indeed meet users' fast-growing demand. But, on the other hand, such flexibility also brings relatively week security mechanisms. Due to the sharing UID design of the underlying virtualization engine, the so-called "shared-everything" threats emerge: a guest app has as many permissions as the host app does, and it can access the data belonging to the host app or other guest apps. It never takes long for malware authors to catch up with the advanced technology trend. According to the latest study from Tencent security lab [61], among all of the apps that are built on VirtualApp, only 8.86% of them are benign, and the others are either malware or the so-called "Potentially Unwanted Programs" (PUPs) [57, 58, 65].

3.1 Threat Model

The possible attack vectors to dual-instance apps come in two ways: 1) malicious host apps; 2) a legitimate host app but with both benign guest apps and malicious guest apps running together. In the first case, malicious APKs are embedded into a host app and will be launched silently when the host app begins to run. As the host app takes complete control of guest apps, many attacks to guest apps such as intercepting API calls or stealing sensitive information become quite straightforward. Recently Google Play Store has removed many DroidPlugin/VirtualApp based apps, and most of them are either malware or PUPs [36, 55, 74]. In the second scenario, attackers trick users into loading malware in a legitimate host app, and then malware conducts attacks when other benign guest apps are running. In addition to social media apps, users also would like to run two copies of a game app so that they can play two characters at the same time. However, repackaging popular Android applications such as game apps has become a common way for malware authors to camouflage malicious code [12, 25, 26, 78]. Therefore, it allows a maliciously repackaged app to snoop on a legitimate guest app. In either case, benign guest apps can be compromised. We perform two case studies to demonstrate the security threats caused by these two attack vectors.

3.2 Case Study: Malicious Host App

Avast security team reported a new malware variant captured from China in 2016 [3], which is believed to be the first malicious host app to launch a phishing attack. It is well known that the Government of China bans some high-ranking websites and apps including Twitter and Facebook[4]. This malware wraps Twitter with a customized version of VirtualApp and sets up a local VPN service

[4]https://en.wikipedia.org/wiki/Websites_blocked_in_mainland_China

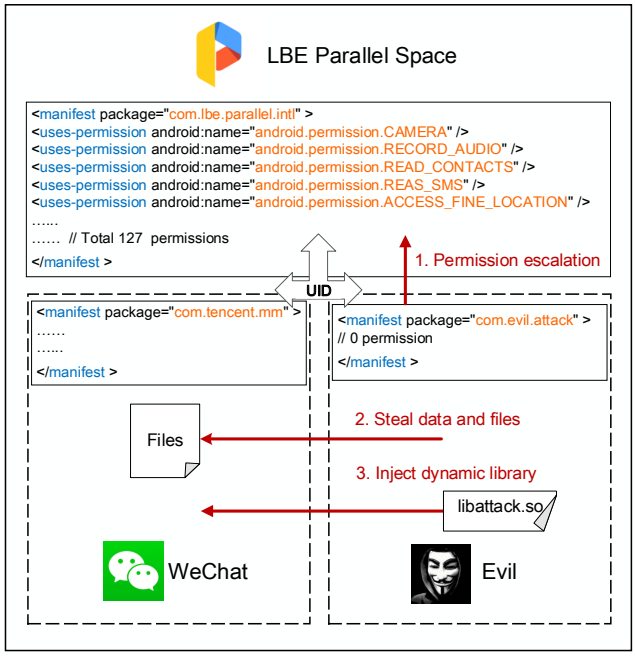

Figure 2: Evil conducts three typical kinds of attacks in dual-instance app context.

to break through the "Great Firewall of China" [15]. In this way, users can directly access Twitter from Mainland China without special VPN configurations. In fact, this malware also loads malicious APKs from its "Assets" directory silently when Twitter is running. It then steals users' Twitter account credentials by hooking the "getText" function of the "EditText" class. To lure users, previous phishing attacks on mobile devices have to overcome several challenges such as developing indistinguishable login GUI [21, 52]. In contrast, application-level virtualization makes phishing attacks much easier as the malicious host app is able to intercept the user's input by nature. Similarly, several malicious host apps target other hot social media apps such as Instagram and WhatsApp [74].

3.3 Case Study: Malicious Guest App

We select LBE Parallel Space, the most popular dual-instance app on Google Play, as the host app. Besides, we choose WeChat as the victim guest app. WeChat is one of the top-ranked instant messaging apps on Android. At last, we develop an app named "Evil" to simulate the possible attacks launched by a malicious guest app. The typical attacking scenario is shown in Figure 2. LBE Parallel Space is installed on an un-rooted Android device, and it loads both WeChat and Evil as guest apps. Evil is able to perform the following three kinds of severe attacks with no permissions or root privilege.

Permission Escalation. Dual-instance apps try to apply for more system permissions to improve compatibility. Evil has *zero* permission, but it can use as many as 127 system permissions applied by LBE Parallel Space at its disposal, including all of the commonly used sensitive permissions. We test Evil's permission escalation by taking sensitive actions such as locating, taking photos, recording,

Table 1: Environment-sensitive app detection results. We test Google Play top 50 apps in three categories: payment/banking, social media, and game.

	Payment/Banking	Social Media	Game
Root Device	31	32	37
Emulator	1	5	9
Dual-instance app	0	0	0

and reading contacts and SMS in mobile phone. Evil succeeds in all cases. LBE Parallel Space fails to provide any security warning when Evil is conducting the behaviors of permission escalation. The new permission model introduced since Android 6.0 allows users to grant a host app's permissions at run time rather than at installation time only. However, malicious guest apps will not stop running even if users deny some permissions of the host app, because they can reuse other available permissions of the host app.

Steal Private Data. When Evil and WeChat are running in parallel under LBE Parallel Space, Evil can get access to WeChat's sensitive files and data, including database and chat records. Due to the permission escalation, Evil does not need WeChat user's authorization.

Hijack Benign App. Evil has the ability to inject malicious code dynamically into other running guest apps. Evil runs the injection tool based on ptrace mechanism [76] to inject the dynamic-link library "libattack.so" into the main process of WeChat. "libattack.so" is then called automatically to intercept "open" function and get the path of record files from its parameter. In this way, it steals audio records. It can even modify the path to send another record file to perform a man-in-the-middle attack.

With the above three attacks together, once a malicious guest app is loaded, it can use dangerous permissions, steal private data, and jeopardize benign apps without root privilege or any other permissions. On the contrary, as the strict access control is not missing in real Android system or normal application-layer sandboxes [5, 9], performing the same attacks to them is difficult, if not impossible.

3.4 Environment-Sensitive Apps

Many critical apps are sensitive to insecure environments: they can detect whether they are running in an emulator or root device. Therefore, a natural question arises: whether they can still detect the risky dual-instance app environments. We test Google Play's top 50 critical apps in three categories: payment/banking, social media, and game. As shown in Table 1, the majority of them can detect the environment of root device, and some apps are able to identify Android emulators such as QEMU [8] or VirtualBox [46]. They will reveal different behaviors when identifying such an insecure environment such as notifying users and then quitting. Surprisingly, none of them embeds the code to detect dual-instance app. That means these critical apps cannot discriminate between application-level virtualization environment and Android system, and traditional fingerprinting features are futile here as well. Our study reveals that most Android developers are not fully aware of the seriousness of security threats raised by dual-instance apps.

4 DEEP VIRTUALAPP INSPECTION

We have demonstrated that running apps in dual-instance app's virtualization environment is at great risk. This section aims to demystify the latest virtualization-based dual-instance apps. Their key characteristics are indispensable to understanding the "shared-everything" threats raised by this new technique and our mitigation. We perform tedious reverse engineering work to study the underlying virtualization mechanism of various dual-instance apps. We sum up the following common characteristics[5].

4.1 Excessive Permissions

Typically, Android apps declare the permissions that they need in the manifest file at installation time, and it is up to users to decide whether to grant the required permissions at run time. The recent statistics show that each benign app applies for 5 permissions on average [11]. As a contrast, due to sharing UID, most host apps have to apply for a plethora of permissions so that they can be compatible with guest apps as much as possible. This also removes the single guest app restriction of NJAS [9]. However, many of these applied permissions are never used by normal guest apps. We analyze all of the 42 dual-instance apps downloaded from four popular Android app stores. According to our study, VirtualApp requires as much as 186 permissions by default, and dual-instance apps apply for 129 permissions on average. Surprisingly, the maximum number is up to 234. Statistically, the permissions required by dual-instance apps range from 90 to 234, and more than 70% dual-instance apps have at least 120 permissions, including the most popular ones. Meanwhile, about 74% of dual-instance apps with more than 120 permissions have over 100K downloads. Obviously, excessive permissions violate the principle of least privilege and may cause permission escalation in guest apps.

4.2 Hooking ClassLoader

This essential step explains where and how to load guest apps without installation, because only host app's dex information is visible to Android system initially. Android apps can load classes, dex files, and APK files from any folder by calling ClassLoader. PathClassLoader and DexClassLoader, which are inherited from BaseDexClassLoader, are two frequently used ClassLoader types. The former only loads dex files, while the latter can load APK, dex, and jar files from an arbitrary directory. DexClassLoader calls a native function *openDexFileNative* to find the path of dex files. During installation, an app's APK file is unzipped into dex files, whose information is saved as dexElement in the DexPathList. As shown in Figure 3, when loading a dex, Android system traverses the dexFileElement list in order until it finds the target. To force Android system to load a guest app, VirtualApp first copies the dex file of guest app to its own directory. Then VirtualApp changes the parameter of *openDexFileNative* by using the native hooking framework, Cydia Substrate [54]. The purpose is to insert the guest app's dex file into the DexPathList ahead of others. Note that VirtualApp patches Cydia Substrate so that it can achieve native hooking without root privilege. In this way, Android system will first match the dexElement of guest app and then load it from VirtualApp's directory.

[5]We take the dual-instance apps that are built on VirtualApp as an example to present.

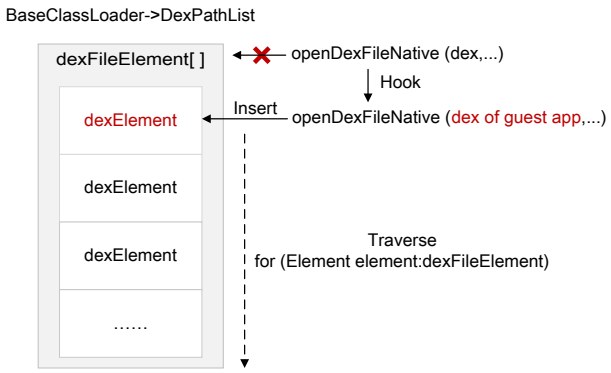

Figure 3: VirtualApp hooks openDexFileNative function to load guest apps without installation.

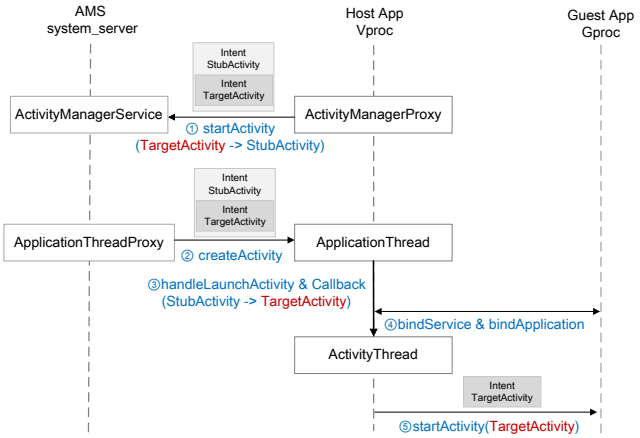

Figure 4: The workflow of starting the Activity of guest app ("TargetActivity"). Two key steps are: 1) predefining dummy components (e.g., "StubActivity") in the manifest file of host app; 2) wrapping/unwrapping "TargetActivity" with "StubActivity" by hooking Android system service APIs (① & ③).

4.3 Component Lifecycle Management

Given loading guest apps without installation, another challenge rears its head: to launch a guest app, the host app must interact with Android system to maintain the lifecycle of guest app components such as Activity, Service, Content Provider, and Broadcast Receiver. However, guest apps' components are not registered in the host app's manifest file beforehand because the host app cannot predicate the specific component names of guest apps. VirtualApp solves this dilemma by predefining dummy components in its manifest file and hooking Android system service APIs. VirtualApp first defines some dummy components and permissions in its own manifest file, including Activity, Service and Content Provider. Android system can certainly maintain the lifecycle of such dummy components. Then VirtualApp utilizes dynamic proxy and reflection techniques to intercept the APIs that are used to manage the lifecycle of components. The purpose is to substitute the target components in guest apps for predefined dummy components. In particular, VirtualApp will modify API parameters or function logic to substitute the BinderProxy of system services such as ActivityManagerService, PackageManagerService, AccountManagerService, and NotificationManagerService.

Taking Activity component as an example, Figure 4 shows how VirtualApp maintains the lifecycle of guest app components. AMS server ("system_server" in Figure 4) manages the task stacks and Activity lifecycle. When launching Activity, application process has to communicate with system_server several rounds through a proxy of AMS (i.e., ActivityManagerProxy). ApplicationThread, which actually is a Binder object, is the bridge between AMS server and application process. After management, system_server will return the control to application process through ApplicationThreadProxy. After that, ApplicationThread will notify ActivityThread to start a new activity. At run time, the host app first intercepts startActivity to wrap the guest app's TargetActivity intent into the intent of StubActivity, which has been predefined in the host app manifest (①). In this way, the host app can deceive AMS server into creating a new activity for StubActivity (②). Then the host app manages to forward TargetActivity rather than StubActivity to ActivityThread. To achieve this, the host app hooks handleLaunchActivity of ApplicationThread class and the callback function of ActivityThread class to extract TargetActivity (③). Meanwhile, the host app process binds service and application with the new guest app process (④). At last, the guest app's Activity component will be launched successfully (⑤). The processes of handling Service, Content Provider, and Broadcast Receiver for guest apps are similar. Additionally, as for Broadcast Receiver, the host app registers static broadcast dynamically.

4.4 Storage Redirection

In general, each app has its own private storage space, and UID property restricts other apps cannot access the private data. However, the strict access control based on different UIDs cannot be applied in dual-instance apps. By means of storage redirection, dual-instance apps use reflection to translate the installation directory of guest apps to physical storage used by the host app. Typically, an app's installation directory is "/data/app/[guest-package]", while the guest app installation directory is changed to " /data/data/[host-package]/virtual/data/app/[guest-package]" or other storage of host app with the hook of native IO functions. Then host app bypasses Android system's access control policy of data isolation and manages all of the guest app data.

5 DUAL-INSTANCE APP FINGERPRINTING FEATURES

To mitigate the "shared-everything" threats, this section investigates how apps can infer whether they are being loaded by a dual-instance app based on the key characteristics of virtualization-based dual-instance apps. We implement our approach as an open-source tool named *DiPrint*.

5.1 Challenges

Fingerprinting Android emulators [8, 14, 29, 46, 64, 67] has been well studied due to its importance in malware defense [22, 27, 34,

Table 2: Dual-instance app fingerprinting features.

Category	Feature
a. Path	1. Host app's APK path in guest app process 2. APK source code and dynamic-link library path
b. UID	3. Multiple processes with the same UID 4. Check undeclared permissions
c. Code Injection & Hooking	5. Stack tracking of exception 6. Suspicious library to provide native hooking

39, 45, 47, 66]. However, our detection target posts a new challenge: dual-instance apps are immune to traditional fingerprinting features. Most discrepancies between Android emulators and real devices do not exist in application-level virtualization. QEMU-based and VirtualBox-based emulators share plenty of hardware-related discrepancies with real systems [34]. The rationale behind is system-level virtualization technology rarely builds a fundamental transparent environment [24]. These discrepancies mainly come from the defects of software-emulated hardware (e.g., Bluetooth, power management, and USB), and they provide a large number of available options for detection heuristics. By contrast, application-level virtualization does not virtualize hardware, so we cannot reuse the previous hardware-related features to detect dual-instance apps.

The second challenge comes from the diversity of virtualization implementation. Apart from VirtualApp and DroidPlugin, 47% of commercial dual-instance apps in our evaluation rely on custom-made virtualization engines. The detection heuristics that only match package name or app signature can be evaded by simple modifications. For example, many commercial dual-instance apps' processes contain the following keywords: "parallel", "clone", or "multi", but our approach does not search them. Instead, we exploit the common design characteristics and configurations.

5.2 Characterizing Fingerprinting Features

To fool Android system, the host app has to patch guest apps in many ways, and a set of guest app's system features will be modified by the host app. Our strategy of selecting detection features is to capture such common modifications with the combination of static and dynamic analysis. Figure 5 shows the workflow of fingerprinting feature generation. Given the static information collected from the manifest and bytecode (e.g., permissions, hook method, and libraries), dynamic analysis performs runtime check and exception tracking to extract runtime discrepancies as fingerprinting features, which are listed in Table 2. We classify them into three categories, and each one contains two features. All of them represent the key characteristics of dual-instance apps that we have summarized in Section 4. Feature 2 and 6 detect dual-instance apps by checking the presence of particular paths or suspicious library. The others (feature 1, 3, 4, and 5) are more robust, and an evasion attempt from them requires the design change of virtualization mechanism.

Category a. Path. The fingerprinting features in this category are related to "Storage Redirection" (Section 4.4). When installing an app in an Android device, Android system copies the original APK to the directory of "/data/app/[package]/" and renames it as "base.apk". Meanwhile, the application data is saved in "/data/data/[package]/". Different applications have their own storage directory to ensure data isolation. However, current dual-instance app design does not meet this security policy. The host app redirects the APK of guest app to the directory of its own when loading guest apps, and then it parses the guest app's APK to get the entry point and component information. Therefore, guest apps will not load resources or code from the default path but from the subdirectory of host app. Note that the host app can access the private directory to copy guest app's APK even without root privilege. The trick to access these files is by calling "getPackageManager().getApplicationInfo(package of guest app, 0).sourceDir". Similar to Linux, the "proc" file system in Android contains a variety of process information such as memory data and network traffic data. From there, DiPrint can get the path of its own code and dynamic-link libraries. In particular, DiPrint will detect the following two path related features.

Feature 1. Host app's APK path in guest app process. Host app loads a guest app after creating a process and preparing the virtual environment for it. In essence, the processes of guest app are created by the host app, which loads the resources of host app in the process memory of guest app during initialization. Hence, we can find two different "base.apk" paths in the process memory: one belongs to the guest app, and the other is the host app. Additionally, Android provides interfaces to view user process memory image based on "proc" file system. According to this observation, DiPrint searches the existence of another different APK path in its own process memory by reading "/proc/self/maps".

Feature 2. APK source code and dynamic-link library path. The apps installed in Android system typically load APK source code from "/data/app/[package]", but most guest apps are loaded from the host app's subdirectories. Similarly, dynamic-link libraries of guest apps are loaded in the same way. DiPrint will load a home-made library, "myLibrary.so", and get the dynamic-link library path from "/proc/self/maps". The home-made library loading path in dual-instance apps is different from the real system. "/data/app/[package]/arm/myLibrary.so" is the typical path in Android system, while the path in dual-instance apps becomes a subdirectory of the host app[6].

Category b. UID. The idea of this category comes from the property of "Excessive Permissions" (Section 4.1). In Android, UID represents the identity of each app during installation, and the processes with the same UID share a same collection of permissions. One of the dual-instance app's key characteristics is that the host app and guest apps have the same identity for Android system and permissions as well. We detect them using the following two features.

Feature 3. Multiple processes with the same UID. Figure 1 shows guest apps share the same UID but different PIDs with the host app. Besides, we find another fact that most host apps will spawn at least two processes: one is user interface process, and the other is service process. As a running guest app knows its package name by nature, with other running process information, the guest app can determine whether there exist additional processes but with the same UID. The APIs "getRunningTasks" and "getRunningAppProcesses" have been deprecated since Android 6.0. We can get process information, including PID and UID, by "ps" command, which is available in all Android versions without any permission.

[6] /data/data/[host-package]/virtual/data/app/[guest-package]/lib/myLibrary.so

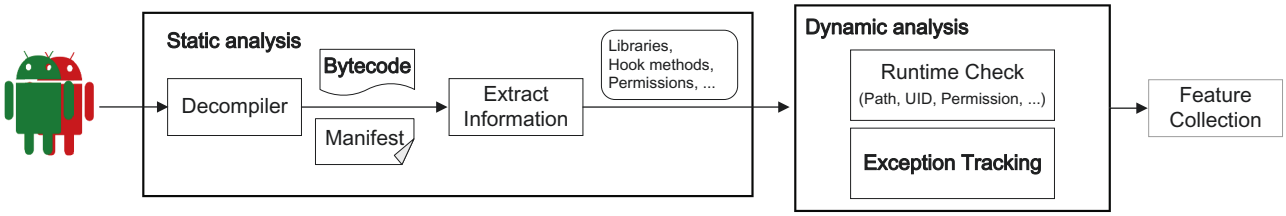

Figure 5: The workflow of fingerprinting feature generation.

Feature 4. Check undeclared permissions. To detect whether guest apps have access to undeclared permissions, DiPrint simulates permission escalation attacks as we discussed in Section 3.3 and check whether it can succeed. DiPrint first collects the required permissions of itself or the apps that embed DiPrint's code, and then it checks the undeclared permissions by calling API "checkCallingOrSelfPermission". At last, it launches a set of actions that require undeclared, dangerous permissions such as READ_SMS, ACCESS_FINE_LOCATION, CAMERA, and RECORD_AUDIO.

Category c. Code Injection & Hooking. In order to deceive both Android system services and guest apps, the host app has to hijack the interactions between them. To achieve this, the host app performs code injection and hooking in several places such as hooking ClassLoader (Section 4.2) and hooking AMS (Section 4.3). The features in this category check the presence of such code injection and hooking.

Feature 5. Stack tracking of exception. Figure 4 shows an example that the host app intercepts the communication between AMS and guest apps. In addition, the host app also creates proxies of other system services. To detect the involvement of host app in the guest app's call chain, our solution is to throw exceptions in the 13 lifecycle functions of 4 components and then analyze the related stack traces. For example, we can throw exceptions in the "onCreate", "onStart", "onResume", "onPause", and "onStop" of Activity. For instance, host apps call their own "callActivityOnCreate" method after calling normal "callActivityOnCreate" method. To find the presence of host app, DiPrint compares the stack trace of exception with the stack trace collected from Android system.

Feature 6. Suspicious library to provide native hooking. To hook native functions, virtualization engines have to customize a dynamic-link library that will be injected into guest apps. For instance, to redirect APK loading path, VirtualApp embeds Substrate framework [54] as "libva++.so", DroidPlugin loads "libsubstrate.so", and MultiDroid[7]. injects "libdaclient.so". According to our study, most dual-instance apps reuse "libiohook.so" and "libjnibridge.so" to provide native hooking, which are the different versions of "libva++.so". In addition, these suspicious libraries are not loaded from the directory of guest app. With the knowledge of loading path and the libraries owned by Android system and DiPrint, we search "/proc/[pid]/maps" file to find the suspicious library that could provide native hooking.

[7]MultiDroid is the underlying virtualization engine of LBE Parallel Space

6 DIPRINT IMPLEMENTATION

To the best of our knowledge, we are the first academic paper to provide a swift response to the emerging threats raised by the popular dual-instance apps. The prototype of our approach, DiPrint, contains two components: fingerprinting feature generation (Figure 5) and runtime dual-instance app detection. The whole tool includes 697 lines of python code, 524 lines of Java code, and 31 lines of C code. In the static analysis of fingerprinting feature generation, DiPrint adopts Apktool v2.3.4 [63] to reverse engineer Android APK files on computer and gets the bytecode and manifest of apps beforehand. To retrieve information of permissions, hook methods, and dynamic-link libraries, we parse the related files with python script. In the dynamic analysis, DiPrint adopts runtime check and exception tracking on-device based on the information extracted from the static analysis. In the runtime dual-instance app detection, developers only need to invoke the detection method by creating an object of DiPrint Class without the modification and root of Android system, which is lightweight and convenient for third-party developers.

```
1  DiPrint diprint = new DiPrint();
2  if (diprint.detect())
3   Notify users or Terminate;
```

Afterward, developers could select different solutions once DiPrint detects the existence of dual-instance app environment, such as alerting users, limiting sensitive functions or terminating execution.

7 EVALUATION

We perform our experiments with several objectives in mind. First and foremost, we want to evaluate whether DiPrint's fingerprinting features are effective to identify dual-instance apps, including both commercial and malicious apps. We also test the false positives when DiPrint is running on various Android devices. After that, we conduct a comparative evaluation to demonstrate the traditional detection heuristics of Android sandboxes do not work on dual-instance apps. At last, we discuss DiPrint's robustness against evasions.

7.1 Experimental Setups

Commercial Dual-Instance Apps. We select commercial dual-instance apps from popular Android application markets and get rid of the crashed ones during running. We attribute these failures to the imperfect implementation of the underlying virtualization engine, which is a sophisticated system program. In addition to Google Play, we also include another three markets (1Mobile Market [1],

Table 3: The distribution of dual-instance apps in four popular Android App stores.

App Store	Google	1Mobile	Tencent	Qihoo	Total
Number	28	4	4	6	42

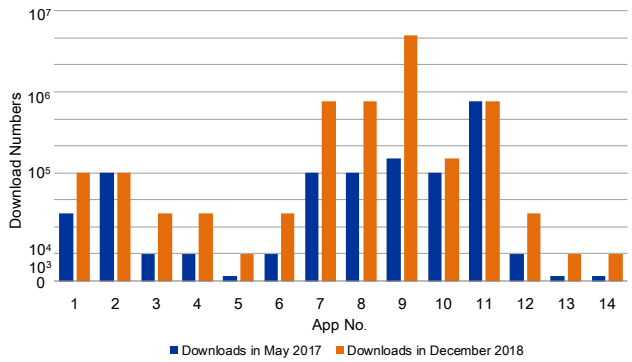

Figure 6: Exponential growth of dual-instance apps from May 2017 to December 2018.

Tencent App Gem [60], and Qihoo 360 Mobile Assistant [49]). Finally, we get a total of 42 dual-instance apps, and the distribution is shown in Table 3. To demonstrate the popularity of dual-instance apps, we compare the download numbers of the apps that are on the market in both May 2017 and December 2018. As shown in Figure 6, we can see an exponential growth for most of them. The downloads of 57% of them increase at least one order of magnitude (e.g., from 500, 000 to 5 million). The most prominent one even bumps up its downloads to a hundredfold.

Real Android Systems. Testin[8] is an app testing service and provides on-demand testing platforms. We test DiPrint on Testin to evaluate DiPrint's false positives over a variety of real Android devices. In total, DiPrint is launched in 109 real systems of different device brands, and the Android versions range from 4.0 to 9.0. Table 4 lists the examples of tested Android devices, which span a wide spectrum of Android versions and brands.

Android Malware. As new Android malware has begun exploiting the innovation of application-level virtualization, we collect top Android malware samples that are active in September 2018 (2, 243 in total) from a leading security company. The only prior knowledge we have is that this malware dataset contains virtualization-based malware. After our experiment, we submit our detection results to that security company and compare with the analysis results from security professionals. As common malware could not load guest apps as virtualization-based malware, we cannot run DiPrint directly on top of them. We leverage Android Asset Packaging Tool to retrieve permission information directly from malware samples and detect the other 5 features dynamically by linking the Android device to a computer with Android Debug Bridge tool. In this way, we could get the running information of malware samples and detect virtualization-based malware.

[8]https://www.testin.net

Table 4: Examples of real Android systems in Testin.

Android Version	Device Number	Device Brand
4.0 - 4.4.4	33	Huawei G620, Redmi 1S, Galaxy Core Max, ...
5.0 - 5.1.1	24	Galaxy Note 3, OPPO A33, Huawei Mate 7, ...
6.0 - 6.0.1	10	vivo Y66L, letv le 2, Nubia Z11, ...
7.0 - 7.1.2	26	Galaxy S7 Edge, vivo X9s, Xiaomi Max 2, ...
8.0 - 8.1.0	14	Huawei P20, OnePlus 3, Lenovo Z5, ...
9.0	2	Pixel 2 XL, Pixel

7.2 Experimental Results

Table 5 shows the experimental results of runtime dual-instance app detection. According to the different underlying virtualization engines, we classify commercial dual-instance apps and Android malware into different categories (Category 1∼7). In summary, DiPrint succeeds in all cases with both zero false positives and zero false negatives. Besides, DiPrint is able to differentiate dual-instance apps from real Android systems and traditional application-layer sandboxes such as Boxify[5] and NJAS[9].

Commercial Dual-Instance Apps. We first reverse engineer the virtualization engines used by commercial dual-instance apps. We find that VirtualApp and DroidPlugin are the most popular options because of their open source. Even so, 47% of them still adopt custom-made virtualization engines. Among all DiPrint's fingerprinting features, Feature 1, 3, 4, and 5 show 100% success rate, which demonstrates their robustness. In spite of this, we still notice the other two features fail in several cases. After further investigation, we find that some host apps do not load guest app resources from their subdirectories but from the original path (e.g., "/data/app/guest-app/"), causing the failure of Feature 2. However, the cost of doing this is the dual-instance copies may crash in the host app environment after the original app updates component information. The reason is the original component information saved in the host app does not match the latest version of APK. In contrast, loading guest apps from the subdirectories of host app (e.g., "/data/data/host-app/data/app/guest-app/") can avoid this update problem by manually copying the updated version to the subdirectories of host app. As for the failure of Feature 6 (suspicious lib to provide native hooking), some dual-instance apps use Java hooking instead of native hooking, which does not need to inject native hooking libraries into the process of guest app.

Virtualization-based Malware. By runtime dynamic detection, we find that about 17.0% of active malware samples in September 2018 are virtualization-based malware. The third-party security company confirmed our detection results. DiPrint's accuracy competes with security professionals but with much smaller overhead for static analysis security company adopts needs more time to parse APKs and analyze code. Compared with security companies, DiPrint detects vulnerable environment dynamically with more timely warnings to users and a more convenient integrating way to developers. For the results of fingerprinting feature hit, we only find

Table 5: The effectiveness experiment of runtime dual-instance app detection. The columns 3~8 show DiPrint's fingerprinting feature (see Table 2) hit numbers. "Y" means that a fingerprinting feature hit, and "N" indicates a fingerprinting feature miss.

Name	Number	1. Exist. host APK	2. APK/Lib path	3. Processes	4. Permissions	5. Stack tracking	6. Hooking lib	Time[1] (ms)	Virtualization Engine
Real Android Systems[2]	109	0	0	0	0	0	0	0.22	None
Commercial Dual-Instance Apps									
Category 1	15	15	15	15	15	15	13	0.41	VirtualApp
Category 2	5	5	3	5	5	5	3	0.28	DroidPlugin
Category 3	2	2	2	2	2	2	2	0.52	MultiDroid
Category 4	20	20	14	20	20	20	13	0.33	Custom-made Engines[3]
Android Malware Samples									
Category 5	294	294	294	294	294	294	147	0.29	VirtualApp
Category 6	63	63	63	63	63	63	42	0.37	DroidPlugin
Category 7	24	24	24	24	24	24	16	0.43	Custom-made Engines
Non-Virtualization-Based Malware	1,862	0	0	0	0	0	0	0.21	None
Representative Examples									
Boxify[5]		Y	Y	N	Y	Y	Y	N/A[4]	
NJAS[9]		Y	Y	N	N	Y	Y	N/A[4]	
com.in.parallel.accounts (GooglePlay)		Y	Y	Y	Y	Y	Y	0.32	VirtualApp
com.ludashi.dualspace (GooglePlay)		Y	N	Y	Y	Y	Y	0.29	VirtualApp
com.qihoo.magic (360)		Y	Y	Y	Y	Y	Y	0.14	DroidPlugin
com.lbe.parallel.intl (GooglePlay)		Y	Y	Y	Y	Y	Y	0.26	MultiDroid
com.youlong.multiaccount (GooglePlay)		Y	N	Y	Y	Y	Y	0.36	Custom-made Engine
com.excelliance.multiaccount (GooglePlay)		Y	Y	Y	Y	Y	Y	0.33	Custom-made Engine
Malware 1: com.twittre.android		Y	Y	Y	Y	Y	Y	0.23	VirtualApp
Malware 2: PluginPhantom		Y	Y	Y	Y	Y	Y	0.41	DroidPlugin
Active Attacker[5]		N	N	N	Y	Y	N	0.81	VirtualApp

[1]We list the average detection time for each dataset.
[2]Real Android Systems: DiPrint's detection results for 109 real Android systems from Testin.
[3]Custom-made Engine is the dual-instance app context framework that is made by the developer and has classical characteristics.
[4]Both Boxify and NJAS are not available to us. We infer the detection results according to their system designs.
[5]Active attackers can adopt possible bypassing ways once DiPrint is known.

some outliers in the feature of native hooking libraries. Another interesting fact is up to 94% of virtualization-based malware rely on VirtualApp or DroidPlugin. Due to the complexity of application-level virtualization engine, its development cost is relatively high. Therefore, malware authors would rather reuse the available solutions to make a quick profit.

We further inspect the identified virtualization-based malware. We find that 59% of them launch phishing attacks and steal user's private data, and the others disguise themselves as legal game apps to hijack payment information when users purchase in-game virtual goods. We submit all of the 2, 243 malware to VirusTotal, and the anti-virus detection rates of virtualization-based malware samples are much lower than other malware. VirusTotal has 60 malware scanning services to detect uploaded samples, but less than 15 malware scanning services are able to label each sample we submit as malware or PUPs. Moreover, no single anti-virus scanner can recognize all of the virtualization-based malware in Category 5~7.

Real Android Systems, Boxify, NJAS, and Representative Examples. The first line of Table 5 summarizes the detection results on 109 real Android devices provided by Testin. DiPrint does not generate any false alarm because none of Android devices matches any DiPrint's fingerprinting feature. We are also interested in knowing DiPrint's results against two traditional application-level virtualization sandboxes, Boxify [5] and NJAS [9]. However, neither of them is available to us. We have to infer DiPrint's results according to their implementation principles. Both Boxify and NJAS reveal the following four similar features with dual-instance apps: 1) the processes of guest apps are created by the host app, which means we can find the existence of host app in the process memory of guest app; 2) a host app also needs to load dex and dynamic-link libraries from its subdirectory or another directory; 3) they intercept system services and the starting process of four components; 4) they use GOT hook or ptrace mechanism to create dynamic-link libraries and inject them to the process of guest app. Therefore, they reveal the same detection results in fingerprinting feature 1, 2, 5, and 6. However, the major difference is that guest apps have

different UIDs with the host app in both Boxify and NJAS (feature 3). Besides, only Boxify's guest app can have excessive permissions, while NJAS allows a customized permission set (feature 4). Also, we present detection details for six top dual-instance apps with high downloads in app markets and two famous virtualization-based malware mentioned in Section 1: one is to steal users' Twitter credentials, and the other can evade the anti-virus scanners.

Overhead. The last but one column of Table 5 shows the detection time. Overall, DiPrint's runtime detection overhead is negligible with an average of 0.36 ms. DiPrint's another component, fingerprinting feature generation, is a one-time effort. It can complete the process of fingerprinting feature generation in 30 seconds.

7.3 Comparison with Traditional Fingerprinting Features

This section tests whether traditional fingerprinting features of dynamic analysis environment can still recognize the virtualization engines of dual-instance apps. It turns out that the fingerprinting features of Android sandboxes are completely ineffective for dual-instance apps.

As Android sandboxes are mainly used for dynamic malware analysis, they need a system-level virtualization solution to restrict the effects of malicious behavior. One fundamental challenge of system-level virtualization is to realistically simulate various hardware effects [24]. That is the reason why most detection heuristics of Android sandboxes attempt to find the hardware-related discrepancies with real systems. DroidAnalyst [23] and Sand-Finger [42] propose a taxonomy of detection methods to check the presence of sandbox environments. We sum up the detection heuristics from these two work into the features shown in Table 6. In addition, we also include the two hypervisor heuristics proposed by Thanasis et al [47]: virtual PC update and cache consistency. They are fairly robust to detect QEMU-based emulators. We compare the results of traditional sandbox fingerprinting features among a sandbox, VirtualApp, and a real Android device. As shown in Table 6, these features can identify the sandbox but fail to distinguish between VirtualApp and a real system.

Mirage [10] presents several dynamic evasion attacks through another common hardware in mobile device, accelerometer. In general, the return values of accelerometer are different between emulators and real devices. We run an emulator and VirtualApp on the same Android device (360 N4S with Android 6.0.1) and check their accelerometer return values. Table 7 shows that, not surprisingly, VirtualApp returns the same results as the real system. It is worth noting that DroidPlugin exhibits the same results with VirtualApp.

7.4 Feature Robustness

Sandbox evasion and anti-evasion [44, 70] are just like a never-ending cat and mouse game. DiPrint is conceptually simple. We have demonstrated its high accuracy and efficiency, but a natural question is: how a skilled attacker can impede DiPrint once our approach is known. This section discusses the robustness of DiPrint's fingerprinting features.

In general, a dual-instance app installs and launches guest apps from its own subdirectory. To bypass Feature 2, host apps could read dex files and dynamic-link libraries directly from the original path of guest apps. However, this evasion is at the cost of stability. When a guest app is loaded for the first time, the host app will cache guest app's information such as Application, Package, and components in order to launch quickly in the next time. If the original app updates with the modification of Application, Package or components, then the APK file (including dex file and dynamic-link libraries) in the original path will be changed as well. As a result, if host apps read APK file directly from the original path of guest apps, the information of the code will be different from the cache saved in host app, which may lead to the crash of guest app. In contrast, if the host app copies the original APK to its subdirectory, it will always load guest app from the code that does not change unless users reinstall the latest version of original app.

Feature 6 is another feature that can be defeated because Java hooking mechanism provides an alternative to native hooking. Nevertheless, the other features are quite robust as they do not mismatch any case in our evaluation. Feature 1, 3 and 4 are the basic design of dual-instance app; Feature 5 comes from the system service proxies used in dual-instance apps. The possible way to evade them requires the design changes of virtualization mechanism. To bypass feature 1, 3 and 5, a determined attacker can hide the trace of hooking or tamper with hooking results with another layer of hooking. Attackers can hook the memory-reading API and process-reading API and return without the host app's information to hide feature 1. Similarly, attackers could bypass feature 3 by hooking the shell-execution API and returning a fake UID that is different from the other app's UIDs. As for feature 5, host app could hide the information of itself by rewriting the Android stack trace when guest app tries to catch exceptions. However, this is never a trivial task considering hooking happens everywhere in dual-instance apps, which always leaves abnormal traces in the stack trace. In addition, because of the shared UID, guest apps could use all common permissions unless users refuse to grant the permissions to host app that will result in blocking the normal functions. DiPrint not only checks the undeclared permissions by calling "checkCallingOrSelfPermission", but also conducts actions that need permissions. Hence, feature 4 cannot be bypassed. The last line of Table 5 represents the worst case that an active attacker can achieve. He succeeds in bypassing four features, but the left two features, permissions and stack tracking, are resilient to evasions.

8 DISCUSSION & FUTURE WORK

Due to the huge demand from mobile users, blocking dual-instance service to avoid the potential risks is a short-sighted solution. We put forward suggestions for both Android developers and users to mitigate this emerging threat. For Android developers, they can integrate DiPrint in their code to remind users that the app is running in an insecure virtualized environment. When normal users are using dual-instance apps, they should not load a large number of apps at the same time. Especially, to prevent privacy leak and property loss, they should not run critical apps such as mobile payment and banking apps in a dual-instance app. Any unidentified or suspicious app cannot be loaded either to avoid malicious guest apps. As malware could hide itself as a dual-instance app to lure users into installing it, mobile phone manufacturers can customize

Table 6: Comparative evaluation with traditional sandbox fingerprinting features. VirtualApp returns the same results as a real Android device.

Category	Features	Sandbox	VirtualApp
a. Hardware	Sensor (Bluetooth, Light Sensor)	N[1]	Y
	Build.HARDWARE	Goldfish/ranchu	Real_Brand[2]
	Build.TAGS	test-keys	release-keys
	Other build properties	"generic", "unknown", "sdk", "Genymotion", ...	N/A[3]
b. Network	IP address	10.0.2/24	Real_IP[2]
	successful Ping	N	Y
	Network interface	eth0	wlan0, sit0, rmnet_data7, lo, ...
c. PhoneID	IMEI	000000000000000	Real_IMEI[2]
	IMSI	310260000000000	Real_IMSI[2]
	Network provider	N	Y
d. File	Existing files	/init.goldfish.rc, /proc/misc, ...	/sys/devices/virtual/switch, /proc_uid_stat, ...
e. Dynamic heuristics	Dynamic accelerometer	stay static	change dynamically
	Power management	stay static	change dynamically
	Running time[4]	less than 10min	more than 10min
	Debugger connected	Y	N
f. QEMU heuristics [47]	Virtual PC update	N	Y
	Cache consistency	N	Y

[1]"Y" indicates that sandbox or VirtualApp does have the feature, and "N" indicates the absence.
[2]Real_brand, Real_IP, Real_IMEI, and Real_IMSI are specific values in different Android devices.
[3]N/A: no keywords.
[4]Real devices often run longer than 10 minutes, but sandboxes are often reset after each analysis.

Table 7: Comparative evaluation of detecting accelerometer return values.

	Emulator	Real System	VirtualApp
getName[1]	Goldfish 3-axis Accelerometer	BMI160	BMI160
getVendor[2]	Android Open sorce project	BOSCH	BOSCH
getFifoMaxEventCount[3]	0	10000	10000

[1]Name of accelerometer.
[2]Vendor of accelerometer.
[3]The max number of events handled in batches by accelerometer.

a benign dual-instance app in their Android systems so that users do not download untrusted third-party tools. Currently, some mobile phone manufacturers, such as Xiaomi and HUAWEI, already have customized OS-level virtualization solutions. Because of the higher privilege, the security of OS-level virtualization is also hard to guarantee. As the principle of application-layer virtualization and its security threats is totally different from OS-level's, we will take it into consideration in future work.

Our approach could detect a dual-instance app effectively, but it can not determine whether the environment is malicious or not. Especially, malware may bypass anti-virus scanners by hiding malicious behavior in multiple guest apps. We plan to study virtualization-based malware in depth and put forward more fine-grained detection features. In addition to dynamic detection, the static analysis of APK file before installation is another choice to defend against virtualization-based malware in advance. We leave it as our future work.

9 CONCLUSION

The popular virtualization-based dual-instance apps have become a "double-edged sword", as they bring new security threats to users. Under dual-instance app context, malicious apps are able to easily elevate privileges, steal sensitive information, and hijack benign apps. Therefore, critical apps such as banking and social networking apps should be aware of being launched by a dual-instance app. In this paper, we first study the security risks of dual-instance apps in depth. To detect dual-instance app at run time, we have studied the common properties of dual-instance app context and extracted effective heuristics as fingerprinting features. The experimental results show that our open source tool, DiPrint, can efficiently detect all dual-instance apps collected from popular Android app markets and virtualization-based malware with zero false positives. Our work exposes the severe security threats raised by dual-instance apps. We do hope dual-instance app developers and security companies will pay more attention to the existing security flaws and the new trend of virtualization-based malware.

10 ACKNOWLEDGMENTS

We sincerely thank our shepherd, Ardalan Amiri Sani, and MobiSys anonymous reviewers for their valuable feedback. We thank Wuhan Antiy Information Technology Co.,Ltd for providing malware samples. This research was supported in part by the National Natural Science Foundation of China(U1636107, 61373168) and the National Science Foundation (NSF) under grant CNS-1850434. Jiang Ming was also supported by UT System STARs Program.

REFERENCES

[1] 1Mobile Inc. last reviewed, 12/10/2018. 1Mobile Market. http://www.1mobile.com/.

[2] asLody. 2015. VirtualApp. https://github.com/asLody/VirtualApp.

[3] Avast Threat Intelligence Team. 2016. Malware posing as dual instance app steals users' Twitter credentials. https://blog.avast.com/malware-posing-as-dual-instance-app-steals-users-twitter-credentials.

[4] Avast Threat Intelligence Team. 2017. Mobile spyware uses sandbox to avoid antivirus detections. https://blog.avast.com/mobile-spyware-uses-sandbox-to-avoid-antivirus-detections.

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

[6] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. 2013. AppGuard - Enforcing User Requirements on Android Apps. In *Proceeding of the 19th international conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'13)*.

[7] Rafael Ballagas, Michael Rohs, Jennifer G Sheridan, and Jan Borchers. 2004. BYOD: Bring Your Own Device. In *Proceedings of the 6th International Conference on Ubiquitous Computing (UbiComp'04)*.

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

[9] Antonio Bianchi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. 2015. 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)*.

[10] Lorenzo Bordoni, Mauro Conti, and Riccardo Spolaor. 2017. Mirage: Toward a Stealthier and Modular Malware Analysis Sandbox for Android. In *Proceedings of the 22th European Symposium on Research in Computer Security (ESORICS'17)*.

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

[12] Kai Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Nan Zhang, Heqing Huang, Wei Zou, and Peng Liu. 2015. Finding Unknown Malice in 10 Seconds: Mass Vetting for New Threats at the Google-play Scale. In *Proceedings of the 24th USENIX Conference on Security Symposium (USENIX Security'15)*.

[13] Wenzhi Chen, Lei Xu, Guoxi Li, and Yang Xiang. 2015. A Lightweight Virtualization Solution for Android Devices. *IEEE Trans. Comput.* 64, 10 (2015), 2741–2751.

[14] Claudio. last reviewed, 12/10/2018. Cuckoo. http://www.cuckoosandbox.org.

[15] Richard Clayton, Steven J. Murdoch, and Robert N. M. Watson. 2006. Ignoring the Great Firewall of China. In *Proceedings of the 6th International Conference on Privacy Enhancing Technologies (PET'06)*.

[16] CtripMobile. 2015. DynamicAPK framework. https://github.com/CtripMobile/DynamicAPK.

[17] Benjamin Davis and Hao Chen. 2013. RetroSkeleton: Retrofitting Android Apps. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services (MobiSys'13)*.

[18] Benjamin Davis, Ben Sanders, Armen Khodaverdian, and Hao Chen. 2012. I-ARM-Droid: A Rewriting Framework for In-App Reference Monitors for Android Applications. *Mobile Security Technologies* 2012, 2 (2012), 17.

[19] Georg Disterer and Carsten Kleiner. 2013. BYOD Bring Your Own Device. *Procedia Technology* 9 (2013).

[20] Dean Evans. 2015. What is BYOD and why is it important? https://www.techradar.com/news/computing/what-is-byod-and-why-is-it-important-1175088.

[21] Adrienne Porter Felt and David Wagner. 2011. Phishing on Mobile Devices. In *Proceedings of the 2011 Web 2.0 Security & Privacy*.

[22] Jyoti Gajrani, Jitendra Sarswat, Meenakshi Tripathi, Vijay Laxmi, Manoj Singh Gaur, and Mauro Conti. 2015. A robust dynamic analysis system preventing SandBox detection by Android malware. In *Proceedings of the 8th International Conference on Security of Information and Networks (SIN'15)*.

[23] Jyoti Gajrani, Jitendra Sarswat, SMeenakshi Tripathi, Vijay Laxmi, M.S. Gaur, and Mauro Conti. 2015. A robust dynamic analysis system preventing SandBox detection by Android malware. In *Proceedings of the 8th International Conference on Security of Information and Networks (SIN'15)*.

[24] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. 2007. Compatibility is Not Transparency: VMM Detection Myths and Realities. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems (HOTOS'07)*.

[25] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. 2012. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys'12)*.

[26] Steve Hanna, Ling Huang, Edward Wu, Saung Li, Charles Chen, and Dawn Song. 2012. Juxtapp: A Scalable System for Detecting Code Reuse Among Android Applications. In *Proceedings of the 9th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'12)*.

[27] Yunfeng Hong, Yongjian Hu, Chun-Ming Lai, S. Felix Wu, Iulian Neamtiu, Patrick McDaniel, Paul Yu, Hasan Cam, and Gail-Joon Ahn. 2017. Defining and Detecting Environment Discrimination in Android Apps. In *SecureComm'17*.

[28] Qihoo Inc. 2015. DroidPlugin. https://github.com/DroidPluginTeam/DroidPlugin.

[29] Israel Levy. 2014. BufferZone. https://bufferzonesecurity.com/.

[30] Dimitry Ivanov. 2016. GOT hook in Android 6.0.1. https://android.googlesource.com/platform/bionic/+/d88e1f350111b3dfd71c6492321f0503cb5540db.

[31] Jinseong Jeon, Kristopher K Micinski, Jeffrey A Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S Foster, and Todd Millstein. 2012. Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices (SPSM'12)*.

[32] Durai Jiiva. 2017. How To Install Same App Multiple Times On Android Device. https://www.premiuminfo.org/install-same-app-multiple-times-android-device/.

[33] Junjie Jin and Wei Zhang. 2017. System Log-Based Android Root State Detection. In *Proceedings of the 3th International Conference on Cloud Computing and Security (ICCCS'17)*.

[34] Yiming Jing, Ziming Zhao, Gail-Joon Ahn, and Hongxin Hu. 2014. Morpheus: Automatically Generating Heuristics to Detect Android Emulators. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC'14)*.

[35] Swati Khandelwal. 2015. How to Run Two WhatsApp Accounts in One Phone. https://thehackernews.com/2015/04/multiple-whatsapp-accounts.html.

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

[37] Taeyeon Ki, Alexander Simeonov, Bhavika Pravin Jain, Chang Min Park, Keshav Sharma, Karthik Dantu, Steven Y. Ko, and Lukasz Ziarek. 2017. Reptor: Enabling API Virtualization on Android for Platform Openness. In *Proceedings of the 15th ACM International Conference on Mobile Systems, Applications, and Services (MobiSys'17)*.

[38] LBE Tech. 2016. 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/.

[39] Yonas Leguesse, Mark Vella, and Joshua Ellul. 2017. AndroNeo: Hardening Android Malware Sandboxes by Predicting Evasion Heuristics. In *Proceedings of the 11th International Conference on Information Security Theory and Practice (WISTP'17)*.

[40] Yi Liu, Yun Ma, and Xuanzhe Liu. 2017. Flexible Installability of Android Apps with App-level Virtualization based Decomposition. *CoRR* abs/1712.00236 (2017).

[41] Nguyen Vu Long, Ngoc Tu Chau, Seongeun Kang, and Souhwan Jung. 2017. Android Rooting: An Arms Race between Evasion and Detection. *Security & Communication Networks* 2017, 3 (2017), 1–13.

[42] Dominik Maier and Mykola Protsenko. 2014. Divide-and-Conquer: Why Android Malware Cannot Be Stopped. In *Proceedings of the 9th International Conference on Availability, Reliability and Security (ARES'14)*.

[43] Shubham Meena. 2017. How To Install Same App Two Times On Android. https://www.trickscity.com/install-one-app-two-times-on-android/.

[44] Najmeh Miramirkhani, Mahathi Priya Appini, Nick Nikiforakis, and Michalis Polychronakis. 2017. Spotless Sandboxes: Evading Malware Analysis Systems Using Wear-and-Tear Artifacts. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P'17)*.

[45] Sebastian Neuner, Victor Van der Veen, Martina Lindorfer, Markus Huber, Georg Merzdovnik, Martin Mulazzani, and Edgar Weippl. 2014. Enter Sandbox: Android Sandbox Comparison. *CoRR* abs/1410.7749 (2014).

[46] Oracle. last reviewed, 12/10/2018. VirtualBox. https://www.virtualbox.org/.

[47] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. 2014. Rage Against the Virtual Machine: Hindering Dynamic Analysis of Android Malware. In *Proceedings of the 7th European Workshop on System Security (EuroSec'14)*.

[48] Dan Price. 2018. How to Run Multiple Copies of the Same App on Android. https://www.makeuseof.com/tag/run-multiple-app-copies-android/.

[49] Qihoo Inc. last reviewed, 12/10/2018. Qihoo 360 Mobile Assistant. http://zhushou.360.cn/.

[50] Siegfried Rasthofer, Steven Arzt, Enrico Lovat, and Eric Bodden. 2014. DroidForce: Enforcing Complex, Data-centric, System-wide Policies in Android. In *Proceeding of the 9th International Conference on Availability, Reliability and Security (ARES'14)*.

[51] Chuangang Ren, Kai Chen, and Peng Liu. 2014. Droidmarking: Resilient Software Watermarking for Impeding Android Application Repackaging. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE'14)*.

[52] Chuangang Ren, Yulong Zhang, Hui Xue, Tao Wei, and Peng Liu. 2015. Towards Discovering and Understanding Task Hijacking in Android. In *Proceedings of the 24th USENIX Conference on Security Symposium (USENIX Security'15)*.

[53] Mitja Rutnik. 2018. What is stock Android? https://www.androidauthority.com/what-is-stock-android-845627/.

[54] SaurikIT. 2014. Cydia Substrate: The powerful code modification platform behind Cydia. http://www.cydiasubstrate.com/.

[55] Rafia Shaikh. 2017. 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/.

[56] Yun Shen, Yun Shen, and Yun Shen. 2015. All your Root Checks are Belong to Us: The Sad State of Root Detection. In *Proceedings of the 13th ACM International Symposium on Mobility Management and Wireless Access (MobiWac'18)*.

[57] Vlasta Stavova, Lenka Dedkova, Vashek Matyas, Mike Just, David Smahel, and Martin Ukrop. 2018. Experimental large-scale review of attractors for detection of potentially unwanted applications. *Computers & Security* 76 (2018), 92–100.

[58] Vlasta Stavova, Vashek Matyas, and Mike Just. 2016. On the impact of warning interfaces for enabling the detection of Potentially Unwanted Applications. In *Proceedings of the 1st European Workshop on Usable Security (EuroUSEC'16)*.

[59] San Tsai Sun, Andrea Cuadros, and Konstantin Beznosov. 2015. Android Rooting: Methods, Detection, and Evasion. In *Proceedings of the 5th ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM'15)*.

[60] Tencent Inc. 2012. Tencent App Gem. http://android.myapp.com/.

[61] Tencent United Security Laboratory. 2018. Tencent TRP-AI anti-virus white paper. https://slab.qq.com/news/authority/1744.html.

[62] JC Torres. 2015. Android Pay won't work with rooted devices, custom ROMS. https://www.slashgear.com/android-pay-wont-work-with-rooted-devices-custom-roms-28406544/.

[63] Connor Tumbleson and Ryszard Wiśniewski. last reviewed, 12/10/2018. Apktool: A tool for reverse engineering Android apk files. https://ibotpeaches.github.io/Apktool/.

[64] Ronen Tzur. last reviewed, 12/10/2018. Sandboxie. https://www.sandboxie.com/.

[65] Tobias Urban, Dennis Tatang, Thorsten Holz, and Norbert Pohlmann. 2018. Towards Understanding Privacy Implications of Adware and Potentially Unwanted Programs. In *Proceedings of the 23rd European Symposium on Research in Computer Security (ESORICS'18)*.

[66] Timothy Vidas and Nicolas Christin. 2014. Evading Android Runtime Analysis via Sandbox Detection. In *Proceedings of the 9th ACM symposium on Information, computer and communications security (ASIACCS'14)*.

[67] VMware. 2014. VMware Workstation. https://www.vmware.com/.

[68] Rubin Xu, Hassen Saïdi, and Ross J Anderson. 2012. Aurasium: Practical Policy Enforcement for Android Applications. In *Proceedings of the 21th USENIX Conference on Security Symposium (USENIX Security'12)*.

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

[70] Akira Yokoyama, Kou Ishii, Rui Tanabe, Yinmin Papa, Katsunari Yoshioka, Tsutomu Matsumoto, Takahiro Kasama, Daisuke Inoue, Michael Brengel, Michael Backes, et al. 2016. SandPrint: Fingerprinting Malware Sandboxes to Provide Intelligence for Sandbox Evasion. In *Proceedings of the 19th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID'16)*.

[71] Wei You, Bin Liang, Wenchang Shi, Shuyang Zhu, Peng Wang, Sikefu Xie, and Xiangyu Zhang. 2016. Reference Hijacking: Patching, Protecting and Analyzing on Unmodified and Non-Rooted Android Devices. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*.

[72] Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu. 2014. 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)*.

[73] Mu Zhang and Heng Yin. 2014. AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications. In *Proceedings of the 21th Network and Distributed System Security Symposium (NDSS'14)*.

[74] Cong Zheng, Wenjun Hu, and Zhi Xu. 2017. 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/.

[75] Cong Zheng and Tongbo Luo. 2016. PluginPhantom: New Android Trojan Abuses "DroidPlugin" Framework. https://researchcenter.paloaltonetworks.com/2016/11/unit42-pluginphantom-new-android-trojan-abuses-droidplugin-framework/.

[76] Min Zheng, Mingshen Sun, and John C.S. Lui. 2014. DroidTrace: A ptrace based Android dynamic analysis system with forward execution capability. In *Proceedings of the 10th International Wireless Communications and Mobile Computing Conference (IWCMC'14)*.

[77] Wu Zhou, Xinwen Zhang, and Xuxian Jiang. 2013. AppInk: Watermarking Android Apps for Repackaging Deterrence. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIACCS'13)*.

[78] Yajin Zhou and Xuxian Jiang. 2012. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P'12)*.