# Parallel Space Traveling: A Security Analysis of App-Level Virtualization in Android

Deshun Dai, Ruixuan Li*, Junwei Tang
School of Computer Science and Technology
Huazhong University of Science and Technology
Wuhan 430074, Hubei, China
{dds,rxli,junweitang}@hust.edu.cn
ds0dai@outlook.com

Ali Davanian, Heng Yin
Department of Computer Science and Engineering
University of California, Riverside
Riverside, CA, United States
{adava003,hengy}@ucr.edu

## ABSTRACT

App-level virtualization becomes increasingly popular. It allows multiple instances of an application to run simultaneously on the same Android system, without requiring modification of the Android firmware. These virtualization-capable apps are used by more than 100 million users worldwide. We conduct a systematic study of the implementation of app-level virtualization and the security threats that their users may face. First, we survey more than 160 apps collected from several popular app markets which can provide application virtualization capability. We find that these apps are implemented based on a similar design, and apps running in such a virtual environment are not completely isolated from each other. Second, we analyze malicious virtualized guest apps, and identify several areas of potential attack vectors, including privilege escalation, code injection, ransomware, etc. Malicious virtualized guest apps can launch reference hijacking attacks. Once a legitimate app is running in the virtual context, all of its sensitive data will be exposed to the host app. Third, we find a new type of repackaging attack. In our collection of 2 million app data set, we find that 68 apps pack and load malwares by using the virtualization technology to evade antivirus detection, 91 apps pack some legal apps for the purpose of wide distribution, and insert screen ads to gain profits at its startup. Finally, we discuss a variety of mitigation solutions for users, developers and vendors.

## KEYWORDS

Mobile security, Android system, application virtualization, security threats, security enhancements

**ACM Reference Format:**
Deshun Dai, Ruixuan Li*, Junwei Tang and Ali Davanian, Heng Yin. 2020. Parallel Space Traveling: A Security Analysis of App-Level Virtualization in Android. In *Proceedings of the 25th ACM Symposium on Access Control Models and Technologies (SACMAT '20), June 10–12, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3381991.3395608

*Corresponding author.


*SACMAT '20, June 10–12, 2020, Barcelona, Spain*

ACM ISBN 978-1-4503-7568-9/20/06.
https://doi.org/10.1145/3381991.3395608

## 1 INTRODUCTION

Android is the most widely used mobile operating system today with the biggest market share of the worldwide smart phone sales volume. The Android smart phone is a personal hand-held device, in which there are a large amount of personal private data. To ensure the user privacy on a native Android system, android system employs sandboxing techniques to protect one application from another. The implementation of the sandboxing is based on mandatory access control of SELinux [15]. Within this sandbox, each application can only access its own files and a handful of system services. This design does not allow execution of multiple instances of an application at the same time.

In some cases, users may want to run multiple instances of an application on the mobile device. For example, a user might have two WeChat (a popular messaging app in China) accounts for personal and business purposes respectively. The user in this scenario doesn't want to switch accounts by logging in and logging out repeatedly, or use two different smart phones. An attractive solution to this problem is application-level virtualization: a virtualization app (HostApp) can load and run other applications (GuestApps) inside the HostApp, due to its convenience and no need for system modification. The most popular virtualization HostApps *(e.g., under package names com.lbe.parallel, com.qihoo.magic, and com.excelliance.dualaid)* claim over 100 million users' downloads. VirtualApp [1] is the most popular app-level virtualization framework published on Github for developers to build and distribute a custom HostApp fastly.

While app-level virtualization is an attractive solution, it is non-trivial to achieve compatibility and security simultaneously. Similar app-level virtualization technologies like Boxify [4] and NJAS [5] provide highly secure sandboxing by placing a virtualized malicious app into an isolated process with zero permissions. However, this design has not been adopted. Instead, to ensure good compatibility (i.e., apps can run properly in the virtual environment), we find most of the existing virtualization-capable apps are implemented with differentiated designs. We inspect their impact on the app isolation mechanism and security enforcement of access controls. Unfortunately, our experiments show that several Android security principles are broken in the existing HostApps. The data of any victim GuestApps can be easily stolen and tampered by another malicious one. Even if some HostApps implemented their own access control strategies, we find that attackers can easily bypass most of them. We also discover a new repackaging attack. Attackers use

[1] https://github.com/asLody/VirtualApp.

open-source virtualization frameworks to pack both legitimate and malicious apps for wide distribution and evading static detection methods. Through virtualization, the repackaging detection based on traditional code similarity comparison will fail. Recently, Zhang et al. [25] study the security problems caused by app virtualization. However, their study does not distinguish whether the problems are caused by HostApps or GuestApps. They study the problem of malicious GuestApps, but do not discuss what types of attacks can be launched by virtualization platform. Moreover, they do not conduct large-scale measurement study to verify the effectiveness of the mitigations. Tongbo et al. [19] propose a detection method based on processes and file directories to identify whether a guest is running in a virtual environment. This method is not effective, since it can easily be hijacked by HostApps to return false information to the guest.

In this paper, we conduct a comprehensive study on app-level virtualization, with respect to its popularity, security threats and potential mitigations. We propose several more effective methods and verify their effectiveness with real HostApps in markets.

In summary, we make the following contributions:

- We study popular open-source virtualization frameworks, and 160 HostApps which have been widely used. We describe, in detail, how app-level virtualization works.
- We systematically study the threats brought by app-level virtualization with respect to both HostApps and GuestApps. We find out that a malicious GuestApp is able to launch a variety of privilege escalation attacks, code injection attacks, ransomeware attacks, and phishing attacks. Further, a malicious HostApp can launch numerous hijacking attacks and launch a novel repackaging attack. We find 68 HostApps indeed package malware using this novel attack method to evade antivirus detection.
- We present a comprehensive discussion on mitigation techniques. For GuestApps, we propose and evaluate several app-level virtualization detection techniques. For HostApps, we propose numerous mitigations at application and system levels.

## 2 MULTI-INSTANCE EXECUTION

### 2.1 Background

An Android application runs in a separate sandboxing environment that isolates data and code execution of one application from another. Android assigns a unique Linux user ID (UID) to an application when it is installed. The UID is used to distinguish the identity of an application, which constitutes the foundation of sandboxing for Android applications. At runtime, the application runs under its assigned UID in a separate process. Based on this UID, Android software stack enforces access control rules that govern the application sandboxing. UID-based sandboxing imposes a limitation on multi-instance execution of an app on Android. A very common challenge rooting in this problem is how to log into two accounts for an app at the same time. To address this challenge, mobile phone vendors and developers begin to explore the multi-instance execution techniques.

There are several techniques for multi-instance app execution. (1) Multi-User Execution. The native Android system that supports

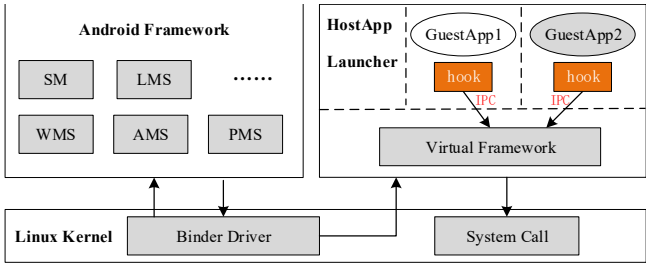


**Figure 1: The overview of application virtualization framework.**

```
1  <activity
2    android:name0="com.lody.virtual.client.stub.
     StubActivity$C0"
3    android:configChanges="mcc|mnc|locale|touchscreen|
     ......"
4    android:process=":p0"
5    android:taskAffinity="com.lody.virtual.vt"
6    android:theme="@style/VATheme"
7  </activity>
```

**Figure 2: The code example for dummy component declared in VirtualApp.**

multiple accounts allows launching different app instances with different accounts. This technique is more reliable and stable. However, switching accounts is expensive and inconvenient. Most mobile phone vendors ban these features. (2) Android System Modification. Some vendors like Samsung, Huawei and Xiaomi modify Android system and provide the function for running multi-instances of an app on their device ROM without switching the system account. Nevertheless, it needs to modify the system, and only supports a few important apps. (3) App ID Modification. The app ID is modified by repackaging, which makes the additional instances of the app run under different IDs. Each instance runs in the sandboxing environment independently, isolated from each other. However, tampering the original app ID value is illegal. Many apps use self-checking to determine whether they are repacked. (4) App-Level Virtualization. A virtualization app (called HostApp) is able to provide a virtualized runtime environment to launch multiple instances of an app (called GuestApp). This technique gains popularity, due to its convenience of use and no need for system modification. Below, we will discuss the technical details of this technique.

### 2.2 App-Level Virtualization

*Android application life cycle.* We briefly describe the life cycle of an android application from installation to execution. Any application before execution needs to be installed. Android Package Manager (PM) parses an application apk file and installs it on the phone. After installation, applications are often loaded by the launcher based on the user UI interactions. The launcher, an Android app, makes an IPC call to the Activity Manager Service (AMS), and drives the AMS to start the user-selected app based on its UID. If the app is not started, the zygote process, a process similar to the

Linux `init`, creates a process with the requested UID, and prepares the codes and data for the process that will be initialized. Finally, the AMS can run the app by simply loading its main activity based on its configuration in the `manifest.xml` file.

*Challenges and goals.* The goal of app-level virtualization is to run an app while being able to control its execution. However, running an app without first installing it is a challenging task. This is because the launcher and activity manager expect the app to be first installed. In addition, the HostApp would like to be able to control the app execution.

*The design.* A HostApp usually has the following components: Launcher; Virtual Framework; and Hook module. Figure 1 depicts how these components interact with the GuestApps and the Android Framework. The Launcher is responsible for the installation, startup, and removal of GuestApps. The Virtual Framework needs to emulate several core system services (e.g., PackageManager, ActivityManager and so on), which facilitate the tasks of managing multiple GuestApps. The Hook module is responsible for intercepting the interaction between the GuestApps and critical system services. Next, we explain how these components achieve the virtualization goal and address the challenges.

The launcher first parses the .apk file and gets the component name of the entry main component that is declared in the manifest file of GuestApp. Note that this is different from the normal application launching as discussed above. Then, the launcher will start a dummy component defined with "process" attribute tag (see in Figure 2). Then the Application process will load the hook module in its initialization process. The hook module communicates with the HostApp through IPC calls. This is how HostApp applies virtualization and controls the key system APIs calls invoked by the GuestApp. The Application process represents the runtime context of the HostApp. Finally, using the Java reflection technology, the value of ActivityThread's field of the bound application object is replaced with that of GuestApp. To wrap up, the new process, the new class loader and the new application context constitute the basic blocks for the GuestApp's execution.

The Virtual Framework relays the communication of the GuestApp with the Android framework. This is essential not only because the HostApp wants to control GuestApp execution (to achieve virtualization goals) but also because the GuestApp execution would fail otherwise. For instance, if we want to start an activity defined in the GuestApp, it leads to a failure in the AMS. This is because the GuestApp's activity is not defined in the host app's manifest file. Note that the effective manifest file is still the HostApp's manifest file not the GuestApp's. The virtual framework solves the problem by predefining several stub components in HostApp's manifest file, and implementing a virtual Activity Manager Service (VAMS) to maintain the life cycle of the app components. Similarly, the Virtual Framework relays communication with other Android Manager services.

The Hook module is the gate to the GuestApp for the HostApp. As mentioned before, the IPC and system calls will be hooked by the Hook Module and sent to the Virtual Framework. The Virtual Framework desires to hook all system call functions associated with file reading and writing (e.g., `open()`, `mkdir()`, `execve()` and so on) through the Hook Module. All of these function calls are encapsulated in the Libc.so dynamic library. The Hook Module can use dynamic library interception technology (e.g., GOT Hook, Inline Hook) within a process to achieve the interception of these functions. For example, when a GuestApp runs in a virtual environment, the Hook Module uses I/O redirection technology to redirect the GuestApp's file directory to a specific file directory allocated in the virtual environment. Finally, a GuestApp's read-write file operations will be relocated to an exclusive sub directory of the HostApp. Therefore, the GuestApp has its own runtime file system to ensure that it can run in the virtual environment.

## 3 SURVEY

We conducted a survey on a wide range of both HostApps and GuestApps. Our survey sheds light on the popularity of apps, the HostApps implementation technology and finally the malice of the apps. We collect a total of about 160 HostApps from several mainstream app markets e.g. Baidu, Yingyongbao, AppChina, Anzhi, 360 and Google Play. We also collect several open-source virtualization frameworks from Github. The analysis of the apps is mostly manual with the support of some existing dynamic and static analysis tools. For example, we use the Apktool [2] to decompile the apk and get the source code. Some samples use packing mechanisms to protect themselves against reverse engineering. For these apps, we use unpacking tools proposed to get the hidden code of these samples. These tools are DexHunter [27], AppSpear [24] and DriodUnpack [10]. In the rest of this section, we report our findings on the aspects outlined above. For each of the findings, we start by a question. We provide a take-home message before providing details on the analysis.

**Which HostApps are popular?**

The most popular HostApp (com.excelliance.dualaid) has been downloaded more than 90 million times. The most popular app-level frameworks are VirtualApp, DroidPlugin and so on. The open source DroidPlugin (5554 stars, 2343 forks in Github) framework is one of the most popular virtualization capable frameworks. From the statistics, we can see that the application virtualization has become an inseparable part in the Android application ecosystem. Many developers use the app virtualization technology to build interesting applications for different needs.

**Which GuestApps are popular?**

We want to know what important applications are loaded and run by the HostApps from the HostApps descriptions. We collect the application descriptions while downloading the HostApps. We analyze the descriptions manually and extract some key words. We find that the largest demands to use the HostApps are social communication apps, such as Twitter, Facebook, WeChat and Weibo, which consists 53.7% of the total uses. Users can log on the app with two different Twitter or WeChat accounts for the personal and business purposes respectively. The second demand is to launch multiple instances of shopping apps. By this way, users can use multiple accounts to get a discount for shopping. These apps store a large amount of user sensitive data.

**How are the HostApps implemented?**

[2] https://ibotpeaches.github.io/Apktool/.

There are different ways to implement app-level virtualization technology (e.g., DroidPlugin, VirtualApp). However, most of them share similar design guidelines as explained in Section 2.2. We find that most of the apps are implemented by reusing existing app virtualization frameworks. We analyze the frequency of the framework reuses, and find that the two most popular open source frameworks are VirtualApp and DroidPlugin, which comprise 67.9% of the share. The result shows that many developers are willing to use open-source virtualization frameworks to develop their apps.

**Are the virtualization applications malicious?**

We conduct an analysis for the samples we collect. We upload 160 samples we collected to VirusTotal [3], and select 20 apps with greater risk and perform static analysis [3] and manual investigation. The results show that most of the HostApps are potentially unwanted programs (PUPs). The capabilities that virtualization technology offers can be used for malicious purposes. Malwares abuse the most popular open source plugin frameworks, "DroidPlugin" and "VirtualApp". Both frameworks can launch arbitrary Android apps without being first installed on the phone. Attackers exploit virtualization technology for evading anti-virus detection, loading malicious programs at runtime, initiating hijacking attacks and launching privilege escalation attacks. In the next section, we elaborate more on the threats imposed by Android application virtualization.

## 4 THREATS

In this section, we present a comprehensive threat analysis for Android application virtualization. The threats that users face originate from either of the two: HostApps or GuestApp. The former is the case when the HostApp is developed by an adversary for malicious purposes. Running a hijacking attack by the HostApp is an example of the possible attacks that we further elaborate on in Section 4.2. The adversary in case of HostApp has more freedom and control over the victim's phone, and the threats that users face are more dangerous.The latter is the case when the HostApp is developed by a trusted source but at least one GuestApp loaded by the HostApp is malicious. The GuestApp is either installed by the user unintentionally, or compromised by an attacker for instance through a malicious third party library. An example of the threats is privilege escalation attack for which we implement a Proof Of Concept (POC). We further elaborate on malicious GuestApp threats in Section 4.1.

### 4.1 Malicious GuestApp

*Threat Model:* There are two security flaws that allow a GuestApp to behave maliciously. First, a GuestApp is not limited to the permissions it defines in its AndroidMainifest.xml file. The HostApp will apply for almost all the permissions and capabilities in advance to support a variety of GuestApps execution. This means that a malicious GuestApp is able to use privileges that are not listed in its AndroidMainifest.xml file but are listed in the host AndroidMainifest.xml file. We analyzed the permissions of more than 160 HostApps we collected from multiple app markets. We counted the number of permissions for each app. The result shows the distribution of permission counts. 84.5% of the apps have 100 or more permissions. Google defines 25 permissions as dangerous permissions [2]. 73.2% of the apps have 20 or more dangerous permissions. Highly privileged applications are more attractive targets for adversaries, and each extra permission extends the attack surface more. Second, a GuestApp can access other GuestApps' private files. In the virtual environment, we find that the various isolation mechanisms advocated by Android systems are broken. We chose more than 10 popular HostApps to test security isolation from three aspects (permissions, storage, components). The test results are shown in Table 1, × indicates that the HostApp does not verify accesses to the sensitive resources, † indicates that the HostApp enforces access controls, but the malicious GuestApp can easily bypass it. For example, *com.excelliance.multiaccounts* enforces access controls by string comparison when GuestApp accesses a specific file or directory. Thus, a malicious GuestApp can use a relative path to bypass it and access the target file of other benign GuestApp. In the following paragraphs, we explain the scenario for each attack.

**Table 1: Vulnerabilities in HostApps**

| Package Name | Permission | Storage | Component |
|---|---|---|---|
| com.excelliance.dualaid | × | × | × |
| com.qihoo.magic | × | † | × |
| com.lbe.parallel | × | × | × |
| com.lbe.parallel.intl | × | † | × |
| com.godinsec.private_space | × | × | × |
| com.ludashi.dualspace | × | × | × |
| com.excelliance.multiaccounts | × | † | × |
| com.parallel.space.lite | × | † | × |
| com.baidu.multiaccount | × | × | × |
| com.morgoo.droidplugin | × | × | × |
| com.lody.virtual | × | × | × |

*Privilege escalation attack:* A malicious GuestApp can perform tasks without declaring the required permissions. We mention that the HostApps apply for many permissions and capabilities in advance to support a variety of functionalities for the GuestApps. For instance, the malicious payload controlled by an attacker is able to access and leak secret data such as user browsing histories and cookies. Further, many apps store valuable data, such as the chat records of WeChat or the login token of Weibo in their private directories. The malicious GuestApp can easily steal all these data.

*Code injection attacks:* A malicious GuestApp can tamper the executable files of another GuestApp. These executable files are loaded via dynamic loading. Many GuestApps may load executable files (e.g., .dex files, .jar files, .so files, etc.) at runtime that are stored in their private directories. The malicious GuestApp is able to tamper or replace these files and hence launch a code injection attack. Our DirDemo can successfully launch this attack because it can write and execute the accessed files.

*Ransomware attacks:* A malicious GuestApp can encrypt and delete another GuestApp's files. Then, the attacker would ask the user for a certain amount of ransom. The users can restore the original files only if they pay the ransom. Especially, some apps like DropBox as a guest app have the ability to automatically propagate files to cloud servers and other client devices. The files encrypted

[3] https://www.virustotal.com.

by malicious GuestApp can be uploaded to the cloud with the Dropbox's auto-sync mechanism.

*Phishing attacks:* The malicious GuestApp can find out what process is running in the foreground, and launch a phishing attack to capture the user sensitive inputs. In the Android versions higher than 5.0, the third-party apps can not get the foreground application process information by calling the *getRunningTasks()* function. However, this restriction is not applied to the virtual environment. More information about phishing attacks is described in paper [17].

*Clone attacks:* When a malicious GuestApp A and normal one B run in the same HostApp, A can secretly pack all runtime files related to B and upload to remote sever. Perhaps an attacker can log into the victim's legitimate app directly without authentication. A popular framework VirtualXposed [4] is a simple app based on VirtualApp that allows users to use an Xposed Module without needing to root, unlock the bootloader, or flash a custom system image. Attackers can easily use this framework to simulate a specific mobile device environment.

## 4.2 Malicious HostApp

*Threat Model:* A malicious HostApp is riskier than a malicious GuestApp. There are two main attacks that a malicious HostApp can launch. The first attack is hijacking in which a HostApp can stop the execution of another application (running outside virtualization) and resume the execution within the virtualization environment under its control. The goal is to hijack some important sensitive information such as input data of users. In the second attack, a HostApp can pack a malware and execute it later as a GuestApp. This is an antivirus evasion approach, and works effectively. The reason is that a malicious HostApp doesn't need any additional functionality to the default ones that a benign one has. In the following paragraphs, we explain each attack in detail and our POC.

*Hijacking:* Several instances of the hijacking attack just explained are found in the wild. Xuan et al. [23] implemented a hijacking attack example called DroidPill. DroidPill enables attackers to hijack the execution of a legitimate application using virtualization they build. Another example is Trojan-Spy.AndroidOS.Twitter that launches Twitter through the integrated VirtualApp framework [20]. After the successful startup of Twitter, the modified VirtualCore module hooks the `getText` function of the `EditText` class. The goal is hijacking user input at the Twitter login window. After the user's login credentials are captured, the malware uploads it to a remote server.

*Antivirus evasion:* Attackers can use the underlying virtualization technology to evade antivirus detection. The application virtualization frameworks provide functionalities that are used by both benign and malicious applications. Henceforth, an antivirus can not raise an alert based on a virtualization functionality. We further elaborate on the techniques that the attackers might use to evade detection using virtualization and the difference with the conventional packing techniques. For example, VirtualApp (VA) is a popular app-level virtualization framework. It allows virtual installation, execution, and uninstallation of arbitrary apps. Note that the apps running in VA do not need to be installed in Android system. Attackers use VA for packing a malware. Alternatively, attackers can load a GuestApp dynamically or fetch it from a remote server at runtime. In this way, the virtualization frameworks doesn't contain any suspicious code.

After packing, the package name, app name and certificate are different from before. Therefore, static detection methods based on package name, certificate or other similar features will fail. When static engines scan suspicious applications, the application information (e.g., package name, certificate, code, etc.) is VA's information that is not malicious. Using virtualization for repacking offers advantages in comparison to the traditional repacking methods. The traditional repacking method is to decompile an application into Java code, modify the class *Application* or *Activity*, embed some ads or malicious payloads, recompile and pack it into a complete apk file. In contrast, by virtualization the attacker does not need to change the original APK for repacking; he can use dynamic code execution through hooking to execute the malicious code. Further, in comparison to the traditional repackaging, detection methods based on code similarity comparison also fails. In the detection process, the code that is compared with the original apk belongs to the HostApp. Obviously, the similarity is not meaningful because the malicious code is not the HostApp itself.

We find several instances of packing using virtualization techniques in the wild. Attackers use these techniques to pack both legitimate apps and malware. We find that about 91 samples pack legitimate apps with virtualization frameworks. In one instance, the attacker uses virtualization framework *com.cx.pluginlib* to pack *air.Stickman* app. The packed app package name is *air.ab.Stickman*. The attackers insert screen ads to gain profits. In another case, we find the same developer using this technology to produce about 73 packed legitimate apps for promotion. We also found about 68 samples that pack malware. We found that 40 apps use VA to wrap an encrypted APK file. Some of these packed malwares are packaged with frameworks such as *com.morgoo.droidplugin* and *com.excelliance.kxqp.platform.gameplugin*. We decrypt the encrypted APK files and upload them to VirusTotal. Almost all the apps are labeled as Riskware, Trojan and Risktool, such as SMSSend and SMSpay.

**Recognizing packaged samples.** The above results are obtained by using the tool developed by ourselves. In the Scanning Process, the first step is to check whether the virtualization technology is integrated in the sample. As we discussed in Section 2.2, the virtual framework will pre-define several stub components in HostApp's manifest file to provide proxies between the GuestApp and the Android framework. These stub components have similar configurations to each other. If more than 80 percent of the components are similar to each other, we consider the corresponding app is very likely has integrated virtualization technology. The second step is to check whether it is repackaged or not. We compare the signature of the inlined files, which are usually stored in the asset directory with the parent apk file, and report it's a packaged sample if the signature of the inlined apk file is different from the parent. However, we cannot deal with some cases if the inlined apk file

[4] https://github.com/android-hacker/VirtualXposed.

was encrypted. We then process it manually, or use VirusTotal to scan the filtered samples and obtain the statistical conclusions.

## 5 MITIGATIONS

The fundamental reason behind all the security flaws is that the GuestApps and the HostApp share the same UID, and hence, the access permissions are shared. In this section, to eliminate the security issues, we describe several possible solutions based on the placement of the solution.

As we discussed in Section 4, the threat might be the HostApp itself or the virtualized GuestApps. Malicious GuestApp can exploit the vulnerabilities of HostApps to attack benign GuestApps, causing privacy leakage, code injection, and other security threats. We also find that benign apps are repackaged by attackers with virtualization frameworks. However, HostApp may also be malicious, once a legitimate app is running in the virtual context, its all sensitive data will be exposed to the host one. To mitigate these security threats, the basic GuestApp-level mitigation is detecting virtualization. Apps may want to prevent execution inside an app-virtualization environment due to risks and lack of enough protection. These mitigation measures or suggestions may be beneficial to app developers. Furthermore, not all developers may want to implement an expensive protection. A robust and lightweight solution defends against these threats is required. We can protect against malicious GuestApps by querying three types of system information that reveal virtualization.

*Detection with private directory info:* The first system info that reveals virtualization is *directory info*. In a virtual environment, the app storage path is different from the real mode. An app's storage path is usually "/data/data/{package name}" in the real mode. However, in the virtualization mode, it is the HostApp that stores the files on behalf of the GuestApp. Hence, each GuestApp will be assigned a subdirectory under the HostApp directory. For example, the dataDir of the GuestApp in Parallel Space is "/data/data//-com.lbe.parallel/parallel/0/{guest app's package name}/data/". Thus, the GuestApp can judge whether it is executed in a virtual environment by invoking the system call `getDataDir()` to obtain location of GuestApp's directory and comparing it with the normal one. However, this detection method is not effective on all HostApps. We found that the HostApp "com.qihoo.magic" can bypass it by returning a directory info similar to the real mode.

*Detection with call stack info:* The second is the *call stack info*. The main idea is that since HostApp (and not the SDK) loads the GuestApp, if the GuestApp finds an external function in the call stack while loading, it is being virtualized. To this end, the GuestApp can add a logic inside the `Application.onCreate()` function or `Activity.onCreate()` to check the call stack. Figure 3 and Figure 4 show the call stack info of a GuestApp running on different platforms. Checking the call stack can be done through the traversing function call chain. Figure 5 shows a sample snippet of the detection code. The logic checks whether there is a function call which is not defined in the Android SDK between two Android Framework API calls. Furthermore, the logic should check whether class loader of this function is not equal to `BootClassLoader`. Note that, the HostApp uses `LoadedApp` class to load the GuestApp.

```
1  java.lang.Throwable
2  at com.example.filetest.MainActivity.onCreate()
3  at android.app.Activity.performCreate()
4  at android.app.Instrumentation.callActivityOnCreate()
5  //at com.lbextern.hook.handle.PluginInstrumentation.
     callActivityOnCreate()
6  at android.app.ActivityThread.performLaunchActivity()
7  at android.app.ActivityThread.handleLaunchActivity()
8  ......
9  at android.app.ActivityThread.main()
```

**Figure 3: Call stack in virtual enviroment.**

```
1  java.lang.Throwable
2  at com.example.filetest.MainActivity.onCreate()
3  at android.app.Activity.performCreate()
4  at android.app.Instrumentation.callActivityOnCreate()
5  at android.app.ActivityThread.performLaunchActivity()
6  at android.app.ActivityThread.handleLaunchActivity()
7  ......
8  at android.app.ActivityThread.main()
```

**Figure 4: Call stack in real Android system.**

```
1 public class MainActivity extends ActionBarActivity {
2   @Override
3   protected void onCreate(Bundle savedInstanceState) {
4       super.onCreate(savedInstanceState);
5       setContentView(R.layout.activity_main);
6      // Get the function call stack list
7       StackTraceElement[] stacks = Thread.currentThread
     ().getStackTrace();
8     // Traversing function call stack chain
9       // and detecting instrumentation
10    for (StackTraceElement st : stacks){
11          String className = st.getClassName();
12          String methodName = st.getMethodName();
13        //...... do checking
14      }
15  }
16 }
```

**Figure 5: A code snippet for determining whether a GuestApp is running in a virtual environment based on the structure of function call stack info.**

*Detection with maps info:* The existing mitigations are not so effective because a malicious HostApp can simply hook the library functions and return false information to mislead the GuestApps. To protect against this, the GuestApp should make direct system calls using native codes. For instance, to retrieve the runtime directory of the GuestApp, we can take the following measure. We can add the logic of obtaining the current process PID in the native layer. Once the value of the PID is obtained, we can find the runtime directory of the current process by reading the memory mapping file, which is stored in directory "/proc/pid/maps". If the runtime directory is different from that of the real mode, the GuestApp should warn the user and stop the execution. In order to avoid neutralization, the GuestApp must obfuscate the codes and use dynamic code execution. See Figure 6 for a detailed code snippet.

```
int checkWithMapsInfo() {
    vector<string> v;
    int pid = (pid_t) syscall(__NR_getpid);
  // Get current process name
    char *process = getCurrentAppProcessName();
    if (process == NULL) return 0;
    size_t len = strlen(process);
    int i = 0;
    // Get the content of current process by reading the
     file '/proc/pid/maps' and store all info in a vector
    getSoMapsPathInfo(pid, "libvenvdetect-lib.so", v);
    for (auto itt = v.begin(); itt != v.end(); itt++) {
        string path = *itt;
        const char *lib = path.c_str();
        if (strstr(lib, process) != NULL) {
      // APP_SO_DATA_APP_PATH = "/data/app/"
      // SO_DATA_APP_LEN = strlen(APP_SO_DATA_APP_PATH)
            if (startsWith(lib, APP_SO_DATA_APP_PATH)) {
                if (strncmp(lib + SO_DATA_APP_LEN,
    process, len)) {
                    i++;
                }
            }
        }
    }
    return i;
}
```

**Figure 6: A code snippet for determining whether a GuestApp is running in a virtual environment based on the structure of maps info.**

**Table 2: Effectiveness of different detection schemes.**

| Package Name | Private Directory Info | Call Stack | Maps Info |
|---|---|---|---|
| com.excelliance.dualaid | ✓ | ✓ | ✓ |
| com.qihoo.magic | × | ✓ | ✓ |
| com.lbe.parallel | ✓ | ✓ | ✓ |
| com.lbe.parallel.intl | ✓ | ✓ | ✓ |
| com.godinsec.private_space | ✓ | ✓ | ✓ |
| com.ludashi.dualspace | ✓ | ✓ | ✓ |
| com.excelliance.multiaccounts | × | ✓ | ✓ |
| com.parallel.space.lite | ✓ | ✓ | ✓ |
| com.baidu.multiaccount | ✓ | ✓ | ✓ |
| com.morgoo.droidplugin | × | ✓ | ✓ |
| com.lody.virtual | ✓ | ✓ | ✓ |

*Evaluation:* We evaluate the effectiveness of the detection schemes mentioned above. The detection results are listed in Table 2, where ticking means effective. We can see that detection using call stack and maps info is more robust than the others.

## 6 DISCUSSIONS

We discuss that how GuestApp detects whether it is being virtualized and then avoid being attacked by a malicious GuestApp or HostApp. However, not all HostApps are malicious. To ensure the security of GuestApps, the developer of legitimate HostApp should play the role of the Android Framework and implement the access control policies. Here, we discuss several possible solutions.

### 6.1 Enhancing the security by modifying HostApps

Under the current implementation, the Android framework can not provide access control for the GuestApps. Therefore, legitimate HostApps must strictly use the blacklisting and signature mechanisms to detect malicious GuestApps. In addition, HostApps must hook sensitive API calls and perform access control before fulfilling GuestApps requests. However, as we mentioned, a GuestApp can subvert this by directly making system calls using native codes. Therefore, a HostApp mitigation must also prevent native system calls. HostApps can do this using *Isolated Process* introduced in Android version 4.1 and higher. An isolated process has fewer privileges than a regular app process. It runs under a separate UID assigned randomly on process startups. This randomly assigned UID differs from any existing UIDs. The sandbox system Boxify is based on Isolated Process [4]. Boxify securely isolates untrusted apps in a completely de-privileged execution environment. Isolated process establishes a strong security boundary between the untrusted GuestApps and Binder IPC reference monitor as they run in separate processes with different UIDs. However, the implementation of a virtualization system based on isolated process is difficult. The restrictions of isolated process may affect the functionality of GuestApps.

### 6.2 Enhancing the security by modifying Android System

Our assumption in the previous subsection was that the HostApp is trustworthy. Otherwise, the HostApp may use the methods we discussed to attack the GuestApp and bypass the GuestApp mitigation; a malicious HostApp can use process isolation to prevent the GuestApp from revealing virtualization. In order to mitigate a malicious HostApp threat, *an explorable solution is to modify the Android system.* The threats can be mitigated by doing access control at the operating system level. The operating system can use runtime information such as call stack and process info to see whether the app is running on the real operating system. If so, there is no need for any further action. Otherwise, the operating system should use package name, UID and PID to build the permission verification mechanism. More specifically, *PackageManagerService* should complete the access control according to a set of policies. For instance, the virtualized apps can call the sensitive APIs only if the policy permits. Alternatively, modify the multi-user mechanism of the operating system to apply multiple instances. The multi-user solution provided by the operating system is definitely more secure than using HostApps developed by third-parties. The other option is blocking app-level virtualization technology partially or outright. App-level virtualization breaks the security model on which Android is based. Android P [5] introduces the restrictions for using non Android SDK interfaces, commonly known as hidden APIs. Android P allows only calling standard interfaces provided by Google whether it is a native or JAVA layer call. This probably breaks the HostApps because almost all virtualization frameworks invoke hidden APIs to implement virtualization.

## 7 RELATED WORK

The related works can be classified into three groups. The first group of works target application virtualization and sandboxing. Related works in this group include three types. The first tries to insert codes into target app's bytecodes for behavior monitoring [1, 8, 9, 12, 28].

[5] https://developer.android.com/preview/restrictions-non-sdk-interfaces.

The second tries to achieve security enhancement by implementing hook on the virtual machine and local libraries [22]. The third uses application virtualization technology and isolated processes for application sandboxing [4, 5, 23]. Instaguard [7] is a new hot patch method allowing immediate deployment for mobile devices.

The second group of works propose solutions to regulate the behaviors of the applications. Smalley et al. [18] extend the SELinux to the Android system, implementing Mandatory Access Control (MAC) on both the kernel and the framework layers. Bugiel et al. [6] use MAC mechanism to mitigate privilege escalation attacks in Android. Roesner et al. [13] propose user-driven access control. This access control grants permissions based on the user actions in the context of an application rather than via manifests or system prompts. There are many related works on the access control and application privilege [11]. Wu et al. [21] analyze the security risks brought by the network open ports in Android applications.

The third group of works propose solutions to regulate the behaviors of the third-party libraries. Shekhar et al. [16] and Zhang et al. [26] run the ad libraries as a separate process so that developers have no need to apply for any permissions for such libraries, which helps restrict the malicious behavior of ad libraries effectively. Seo et al. [14] extend Android permission system by providing in-app privilege separation and develop a novel security mechanism, called inter-process stack inspection that is effective to isolate third-party libraries' permissions from their host apps. Meanwhile, they leverage Hardware Fault Isolation approach to implement native sandboxing and confine memory access strictly by executing JNI in the restricted memory domain.

## 8 CONCLUSION

In this paper, we conducted a systematic study of the implementation of app-level virtualization and the security threats that will be faced by users. We investigated a batch of virtualization-capable applications developed by third parties. We find that these apps are very popular among users and implemented with the same design logic. They can load the GuestApp running in an environment that is not completely isolated from each other. Hence, for malicious virtualized apps or integrated third-party libraries, we revealed several potential attacks (privilege escalation, code injection, ransomware, etc.). We also found the virtualization technology has been abused by malwares which can launch several new attacks. The attacker can implement hijacking attacks without carrying any phishing code or repackage malware inside virtualization frameworks for evading antivirus detection. We find 159 samples with repackaging from several app markets. 68 of them packed malware for wide distribution. Finally, we discussed a variety of mitigation solutions for users, developers and vendors.

## 9 ACKNOWLEDGMENTS

This work is supported by the National Key Research and Development Program of China under grants 2016YFB0800402 and 2016QY01W0202, National Natural Science Foundation of China under grants U1836204, U1936108, 61572221, 61433006, U1401258, 61572222 and 61502185, and Major Projects of the National Social Science Foundation under grant 16ZDA092.

## REFERENCES

[1] Vitor Afonso, Antonio Bianchi, Yanick Fratantonio, Adam DoupÃĬ, and Giovanni Vigna. 2016. Going Native: Using a Large-Scale Analysis of Android Apps to Create a Practical Native-Code Sandboxing Policy. In *NDSS 2016*.
[2] Android 2018. Permissions. (2018). https://developer.android.com/guide/topics/permissions/overview.
[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick Mcdaniel. 2014. FlowDroid:precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.
[4] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp Von Styp-Rekowsky. 2015. Boxify: full-fledged app sandboxing for stock android. In *Usenix Security Symposium*. 691–706.
[5] Antonio Bianchi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. 2015. NJAS: Sandboxing Unmodified Applications in non-rooted Devices Running stock Android. In *ACM CCS Workshop on SPSM 2015*. 27–38.
[6] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad Reza Sadeghi, and Bhargava Shastry. 2012. Towards Taming Privilege-Escalation Attacks on Android. In *NDSS 2012*. 346–360.
[7] Yaohui Chen, Yuping Li, Lu Long, Yueh Hsun Lin, and Xinming Ou. 2018. InstaGuard: Instantly Deployable Hot-patches for Vulnerable System Programs on Android. In *NDSS 2018*.
[8] Benjamin Davis and Hao Chen. 2013. RetroSkeleton: retrofitting android apps. In *ACM MobiSys 2013*. 181–192.
[9] Benjamin Davis, Ben Sanders, Armen Khodaverdian, and Hao Chen. 2012. I-ARM-Droid: A Rewriting Framework for In-App Reference Monitors for Android Applications. In *MoST 2012*.
[10] Yue Duan, Mu Zhang, Abhishek Vasisht Bhaskar, Heng Yin, Xiaorui Pan, Tongxin Li, Xueqiang Wang, and Xiao Feng Wang. 2018. Things You May Not Know About Android (Un)Packers: A Systematic Study based on Whole-System Emulation. In *NDSS 2018*.
[11] Tuncay GÃijliz Seray, Demetriou Soteris, Ganju Karan, and A. Gunter Carl. 2018. Resolving the Predicament of Android Custom Permissions. *NDSS 2018*.
[12] 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 *ACM Workshop on SPSM 2012*. 3–14.
[13] Franziska Roesner, Tadayoshi Kohno, Alexander Moshchuk, Bryan Parno, Helen J. Wang, and Crispin Cowan. 2012. User-Driven Access Control: Rethinking Permission Granting in Modern Operating Systems. In *IEEE Symposium on Security and Privacy*. 224–238.
[14] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Taesoo Kim, Insik Shin, Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Taesoo Kim, and Insik Shin. 2016. FLEXDROID: Enforcing In-App Privilege Separation in Android. In *NDSS 2016*.
[15] Asaf Shabtai, Yuval Fledel, and Yuval Elovici. 2010. Securing Android-Powered Mobile Devices Using SELinux. *IEEE Security & Privacy* 8, 3 (2010), 36–44.
[16] Shashi Shekhar, Michael Dietz, and Dan S Wallach. 2012. AdSplit: Separating smartphone advertising from applications. In *Usenix Security Symposium*. 99.
[17] Aonzo Simone, Merlo Alessio, Tavella Giulio, and Fratantonio Yanick. 2018. Phishing Attacks on Modern Android.. In *ACM CCS 2018*. 1788–1801.
[18] Stephen Smalley and Robert Craig. 2013. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *NDSS 2013*.
[19] Luo Tongbo, Zheng Cong, Xu Zhi, and Ouyang Xin. 2017. Anti-plugin: DonâĂŹt Let Your App Play As an Android Plugin. In *Blackhat Asia 2017*.
[20] TrojanSpy 2017. Dual Instance. (2017). https://blog.avast.com/malware-posing-as-dual-instance-app-steals-users-twitter-credentials.
[21] Daoyuan Wu, Debin Gao, Rocky K. C. Chang, En He, Eric K. T. Cheng, and Robert H. Deng. 2019. Understanding Open Ports in Android Applications: Discovery, Diagnosis, and Security Assessment. In *NDSS 2019*.
[22] Rubin Xu and Ross Anderson. 2012. Aurasium: practical policy enforcement for Android applications. In *Usenix Security Symposium*. 27.
[23] Chaoting Xuan, Gong Chen, and Erich Stuntebeck. 2017. DroidPill: Pwn Your Daily-Use Apps. In *ACM AsiaCCS 2017*. 678–689.
[24] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Junliang Shu, Bodong Li, Wenjun Hu, and Dawu Gu. 2015. AppSpear: Bytecode Decrypting and DEX Reassembling for Packed Android Malware. In *RAID 2015*. 359–381.
[25] Lei Zhang, Zhemin Yang, Yuyu He, Mingqi Li, Sen Yang, Min Yang, Yuan Zhang, and Zhiyun Qian. 2019. App in the Middle: Demystify Application Virtualization in Android and its Security Threats. *POMACS* 3, 1 (2019), 17:1–17:24.
[26] Xiao Zhang, Amit Ahlawat, and Wenliang Du. 2013. AFrame: isolating advertisements from mobile applications in Android. In *ACSAC 2013*. 9–18.
[27] Yueqian Zhang, Xiapu Luo, and Haoyang Yin. 2015. DexHunter: Toward Extracting Hidden Code from Packed Android Applications. In *ESORICS 2015*. 293–311.
[28] Yajin Zhou, Kunal Patel, Lei Wu, Zhi Wang, and Xuxian Jiang. 2015. Hybrid User-level Sandboxing of Third-party Android Apps. In *ACM AsiaCCS 2015*. 19–30.