App in the Middle: Demystify Application Virtualization in Android and its Security Threats

LEI ZHANG, Fudan University, China
ZHEMIN YANG, Fudan University, China
YUYU HE, Fudan University, China
MINGQI LI, Fudan University, China
SEN YANG, Fudan University, China
MIN YANG, Fudan University, China
YUAN ZHANG, Fudan University, China
ZHIYUN QIAN, University of California Riverside, USA

Customizability is a key feature of the Android operating system that differentiates it from Apple's iOS. One concrete feature that gaining popularity is called "app virtualization". This feature allows multiple copies of the same app to be installed and opened simultaneously (e.g., with multiple accounts logged in). Virtualization frameworks are used by more than 100 million users worldwide. As with any new system features, we are interested in two aspects: (1) whether the feature itself introduces security risks and (2) whether the feature is abused for unintended purposes. This paper conducts a systematic study on the two aspects of the app virtualization techniques.

With a thorough study of 32 popular virtualization frameworks from Google Play, we identify seven areas of potential attack vectors and find that most of the frameworks are susceptible to them. By deeply investigating their ecosystem, we show, with demonstrations, that attackers can easily distribute malware that takes advantage of these attack vectors. In addition, we show that the same virtualization techniques are also abused by malware as an alternative and easy-to-use repackaging mechanism. To this end, we design and implement a new app repackage detector. After scanning 250,145 apps from app markets, it finds 164 repackaged apps that attempt to steal user credentials and private data.

CCS Concepts: • **Security and privacy → Software and application security**; *Mobile platform security*; *Virtualization and security*.

Additional Key Words and Phrases: Application Virtualization, Access Control, Android Security

ACM Reference Format:
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 . *Proc. ACM Meas. Anal. Comput. Syst.* 3, 1, Article 17 (March 2019), 24 pages. https://doi.org/10.1145/3311088

Authors' addresses: Lei Zhang, Fudan University, Shanghai, China, lei_zhang14@fudan.edu.cn; Zhemin Yang, Fudan University, Shanghai, China, yangzhemin@fudan.edu.cn; Yuyu He, Fudan University, Shanghai, China, heyy16@fudan.edu.cn; Mingqi Li, Fudan University, Shanghai, China, limq16@fudan.edu.cn; Sen Yang, Fudan University, Shanghai, China, syang15@fudan.edu.cn; Min Yang, Fudan University, Shanghai, China, m_yang@fudan.edu.cn; Yuan Zhang, Fudan University, Shanghai, China, yuanxzhang@fudan.edu.cn; Zhiyun Qian, University of California Riverside, California Riverside, USA, zhiyunq@cs.ucr.edu.

2476-1249/2019/3-ART17 $15.00
https://doi.org/10.1145/3311088

1 INTRODUCTION

As a popular operating system, Android has 2 billion [3] monthly active devices around the world with a huge number of applications (apps for short) that are connected to our daily life, including social network, games and digital wallets. Android users tend to customize the systems and apps — something that Android offers as a major differentiation factor from Apple. For example, users may want to replace the default theme of Twitter, to mock the device location when playing the game Pokemon Go, or to manage multiple accounts of WhatsApp simultaneously. However, many apps apply obfuscation/packing techniques to prevent themselves from being modified/repackaged.

The demand of app customization forms the basis of the growing popularity of app virtualization in Android, which provides mobile users the capability to easily customize obfuscated/packed Android apps. Specifically, a virtualization framework (an Android app) implements a virtual execution environment, and other apps can be executed on top of it. Then, the framework provides the ability to instrument and modify the virtualized apps. One of the most popular virtualization frameworks, Parallel Space, claims that over 100 million users worldwide are using the app [33]. Besides, considering only Google Play, dozens of virtualization frameworks are available to the public, and they are already downloaded by over 84 million users. As many sensitive apps can be run on top of these virtualization frameworks, the security of these frameworks is critical. Prior work [44] reported that app virtualization techniques can be abused to launch several attacks. Specifically, two attacks leverage the vulnerabilities of app virtualization frameworks, and the remaining one is a malware that leverages app virtualization to evade the malware detectors. However, since there is a lack of systematic studies of the vulnerabilities in the wild and a deep understanding of the attack surface, the severity of the threat is still unclear.

We first investigate the security mechanisms built into these frameworks. Specifically, app virtualization can be viewed as a new layer of sandboxing infrastructure where apps running on top should be isolated from each other, e.g., one app can not steal data from another. However, app virtualization inherently changes the assumption of the Android's original sandboxing mechanism. That is, *from the Android operating system's perspective, it fails to identify different virtualized apps executed in the same virtualization framework*. It is the app virtualization framework's responsibility to further isolate the apps running on top. We empirically analyze the 32 popular frameworks we collected from Google Play regarding their app isolation mechanisms and security enforcement of access controls. We are surprised to find that the content of any app can be easily stolen/tampered by another. We further study these frameworks by probing whether the Android security considerations remain enforced. Unfortunately, by studying eight Android security considerations, our experiments show that almost all the access control mechanisms are broken in all the virtualization frameworks. Although some frameworks attempt to remediate the broken sandboxing assumptions, we find that attackers can bypass most of them.

Additionally, we study the severeness of these vulnerabilities. Specifically, (1) what apps can be attacked? (2) how can malware be distributed to end-users? By analyzing user reviews of virtualization frameworks on Google Play, we observed that the users mostly utilize these frameworks to customize social apps (about 56.7%) and games (about 16.6%). Thus, a lot of user privacy (managed by social apps) are potentially exposed to malware that run on top of the same virtualization framework. Then, we study how users can launch and use Android apps in the virtualization frameworks. We observed that almost all these frameworks support app installations from insecure sources. For example, users can install apk files from globally accessible SD cards, which can be easily tempered with by any app on the same device. Furthermore, some commercial virtualization apps provide embedded app markets with insecure transmission channels. Additionally, a common usage of virtualization frameworks for gaming is to use bots, and game bots are not available on the app markets [29]. Thus, to install bots, users download and install apps from non-official and insecure sources (e.g., 3rd party game forums). We study a well-known game bots community which utilizes the virtualization frameworks. Interestingly, as we will describe in §4.2.2, we find that to instrument game apps, the bots framework is utilizing a vulnerability introduced in this paper. We also illustrate that developers can easily distribute malicious game bots which target virtualization frameworks. An exploitation demo (https://youtu.be/Mk_ZISSitow) is provided to illustrate this attack scenario. Additionally, we observed that an app on Google Play attempts to use one of the vulnerabilities discussed in this paper (details in §7) and we argue that these vulnerabilities may cause severe consequences in the future.

Another important but orthogonal question is whether the app virtualization framework itself can be abused to achieve unintended functionality. For example, we observe that app virtualization is used by developers as an alternative approach to repackage existing Android apps. Repackaging apps used to be a common way to create and distribute malware [39] — making a malicious version of Angry Birds with a backdoor. As defenses, a large number of detection techniques (mostly based on similarity) were proposed to detect app repackaging [21]. However, using app virtualization, existing techniques become insufficient.One reason is that the original app is now wrapped by the virtualization framework, which encrypts the original app and decrypts it at runtime, and is invisible to traditional static analysis tools. We therefore propose a new approach to address this challenge. Specifically, by analyzing 250,145 apps automatically from four app markets, we are able to locate 164 zero-day malware and 29 suspicious apps. Furthermore, we upload these apps to VirusTotal [37], and find that although some apps are labeled as PUP (potential unwanted program), few of them are explicitly labeled as malware. Our further investigation shows that the existing anti-virus engines cannot tell malware from benign app virtualization frameworks.

Contributions. The contributions of our work are summarized as follows:

- We systematically identified the security vulnerabilities of app virtualization, and revealed that many Android existing security assumptions/policies are broken in almost all studied virtualization frameworks, causing serious consequences (e.g., stolen cookie and login token).
- We thoroughly investigated the severeness of these vulnerabilities. Our experiments revealed that a large amount of private data (in social apps) are exposed to malware, and malware can be easily pushed to users who use the virtualization frameworks.
- We pointed out that malware is using app virtualization as an alternative way to repackage apps. We proposed a new approach to detect such malware and find 164 zero-day Android malware in 4 app markets.

2 BACKGROUND

In this section, we provide necessary background for understanding how the app virtualization works and how the access controls are performed in Android.

2.1 App virtualization

App customizations are highly demanded by Android users. For example, many users need to manage multiple Facebook or Twitter accounts within one Android device. However, the social apps normally disallow the users to login multiple accounts simultaneously from the same device. Besides, many apps apply obfuscation or packing techniques to protect their code integrity. Thus, it is difficult to modify a highly obfuscated Android app. App virtualization is a popular solution to customize apps without modifying their code.

Currently, app virtualization is applied to achieve various customization requirements, including: (1) creating multiple instances of a customized app without installing it to an Android device;

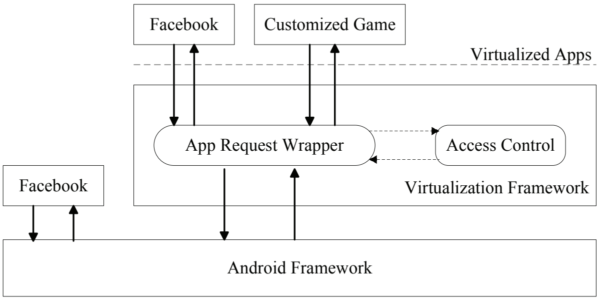

Fig. 1. The overall architecture of virtualization frameworks.

(2) mocking device information including IMEI, phone number, location, etc; (3) bots, for example, automatically clicking the window of a game, sending messages to a group of people; (4) encrypting/decrypting user privacy in a given app; (5) breaching device restrictions. For example, in China, Google Play Service (gps) is forbidden, thus most of the Android devices cannot utilize the gps framework, causing many apps to be not functional. Android users can break the restriction by using a virtualization framework which pre-installed a gps framework.

As depicted in Figure 1, virtualization frameworks act as a virtual execution environment. Specifically, they load the code of the customized apps at runtime, and execute the code in their own process. When an app is executed on such a framework, all its requests to Android are collected and submitted by the request wrappers. On one hand, these wrappers are convenient to customize the virtualized apps, for example, to forge the location by modifying the response of system service *LocationManagerService*. On the other hand, they take the place of the virtualized apps to communicate with the Android framework. As a result, Android determines the virtualization framework as the owner of all the requests, although none of them are generated by the framework itself. However, many Android access control policies are built upon the assumption that Android can correctly identify the senders of requests, which is invalid in the virtualization frameworks. For example, to enforce that each app can only manipulate its own internal storage, Android framework checks the uid of apps. However, for all apps executed on the same virtualization framework, their code is executed under the same uid, that is, the uid of the framework. As a result, the isolation mechanism is broken. Actually, in any of the studied virtualization frameworks, a virtualized app can arbitrarily read or write files of another. Thus, a systematic review of Android security model in virtualization frameworks is necessary.

2.2 Access controls in Android

To maintain the security of Android system and mobile users, Android enforces many access control mechanisms, including its permission-based security model and some app-level isolation policies.

Permissions. To prevent apps from abusing Android system resources, Android provides a set of permissions and ensures that each app should be granted the corresponding permission before it accesses a sensitive resource. Before Android version 6.0, permissions can only be granted when users are installing apps. To install an app, users should grant all the requested permissions to the installed app, otherwise the installation is canceled. After version 6.0, Android starts to support dynamic permission granting. In the new permission model, *normal* level permissions are granted automatically, and any higher-level permissions should be explicitly granted. Specifically, when an app is going to access a system resource with a *dangerous* level permission, a popup window appears to ask the user to grant the permission or deny the access. Besides, *signature* level permissions can only be granted to apps signed by the same developer.

App-level Isolations. Android security model provides a set of isolation and access control policies. As a fundamental piece of this model, every Android app is assigned a unique id (*uid*). Then, Android isolates the execution spaces of apps by their uids. For example, Figure 2 illustrates a code snippet, which verifies the identity of the caller by checking its uid.

```
int uid = Binder.getCallingUid();
if (uid == System_UID){
          #  sensitive operation
} else {
          throw new SecurityException(... ...);
}
```

Fig. 2. Code example for enforcing app-level isolations by checking the caller uid.

3 VULNERABILITIES OF VIRTUALIZATION FRAMEWORKS

As we introduced in §2.1, the adoption of app virtualization may break many security assumptions of the Android access control system, causing security threats to the apps executed on the virtualization frameworks. In this section, we first give an example to illustrate the real-world damage of vulnerabilities in these frameworks. Then, we systematically study the virtualization frameworks collected from Google Play.

Our results in §3.5 show various security threats to these frameworks. Finally, we show several real-world exploitation demos and introduce how they work. We reported these vulnerabilities to the corresponding developers of the virtualization frameworks, and now we received confirmation from one of them.

3.1 A motivating example

WhatsApp is one of the most popular social apps in the world, and it maintains a plenty of user privacy. To manage two WhatsApp accounts simultaneously in one Android device, many users install it into a virtualization framework, e.g., Parallel Space. By calling *getDataDir()*, WhatsApp aims to get the directory of its internal storage(*/data/data/WhatsApp/* in Android file system). However, to achieve virtualization, Parallel Space redirects the return value of this function call to a specific subdirectory of its own storage (*/data/data/Parallel_Space/parallel/0/WhatsApp/*). As a result, all user private data are stored to the new location. Suppose a malware Mal is also installed to the virtualization framework. When Mal is attempting to access */data/data/Parallel_Space/parallel/0/WhatsApp/*, Android security policy is not violated because both WhatsApp and Mal are executed in the same app (Parallel Space) with the same uid. Furthermore, since Parallel Space does not restrict the access of Mal, user privacy managed by WhatsApp is leaked to Mal. Our proof-of-concept demo illustrates this attack.

3.2 Overall methodology

Android implements a series of isolation and access control policies to restrict the capability of Android apps. However, our motivating example illustrates that apps running on commercial virtualization frameworks can easily bypass certain security policies. To thoroughly unveil the security of these virtualization frameworks, we summarize the security concerns enforced by the

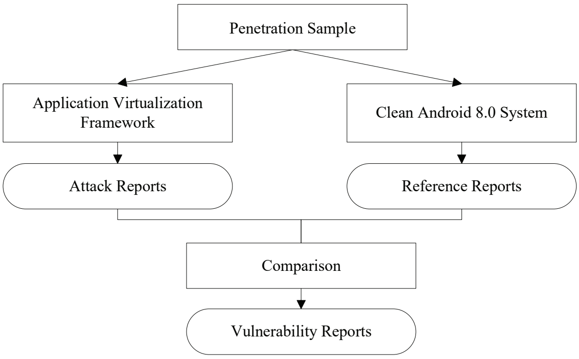

Fig. 3. The overall architecture of our methodology for discovering vulnerabilities in app virtualization frameworks. We use a clean Android 8.0 system as reference, since it is currently the newest mainstream version of Android, in which almost all the security enforcement for Android are applied.

Android operating system, and design a set of penetration test samples that carry a set of malicious behaviors. Then, we run the samples on various virtualization frameworks. We also execute the sample on a clean Android system as a reference. If a malicious behavior fails in the reference system but succeeds in a virtualization framework, we report that the framework is vulnerable to the corresponding behavior. Figure 3 reveals our methodology to systematically study the security of virtualization frameworks.

3.3 Studied virtualization frameworks

Our crawling process that collects virtualization frameworks works as follows: first, we locate a popular virtualization framework (Parallel Space). Then, we search this app in Google Play, where a list of apps are reported as similar apps. We manually check them and identify new virtualization frameworks. By repeatedly searching for similar apps, we successfully locate 32 app virtualization frameworks. We list the ten most popular virtualization frameworks in Table 1.

Package Name	#Downloads
com.lbe.parallel.intl	50,000,000+
com.ludashi.dualspace	10,000,000+
info.cloneapp.mochat.in.goast	10,000,000+
com.parallel.space.lite	5,000,000+
com.jiubang.commerce.gomultiple	5,000,000+
com.excelliance.multiaccounts	1,000,000+
com.ludashi.superboost	500,000+
com.in.parallel.accounts	500,000+
com.nox.mopen.app	500,000+
com.polestar.domultiple	100,000+

Table 1. Parts of the virtualization frameworks studied in this paper. All these frameworks are downloaded from Google Play. The second column lists their download numbers on Google Play.

3.4 Penetration test configuration

Permissions. As aforementioned, permissions are granted to apps by the Android framework. However, when using app virtualization, all the virtualized apps are executed on the virtualization framework, instead of the Android system. Thus, no matter how many virtualized apps are executed, the only app installed to Android devices is the framework itself. As a result, all the virtualized apps can inherit the permissions granted to the virtualization framework. To prevent apps from abusing privileged system resources, virtualization frameworks should implement Android permission model themselves. Our experiments evaluate whether an app with no permission can access a permission protected resource. Specifically, our penetration test sample attempts to access the precise device location (which is protected by a dangerous permission *ACCESS_FINE_GRAINED_LOCATION*) without explicitly granted permissions. It also runs a background service and stealthily sends an SMS message without requesting for the corresponding permission *android.permission.SEND_SMS*. Furthermore, our sample attempts to access the resources in the Amazon app, which is protected by a signature level 3rd party permission (*com.amazon.CONTENT_PROVIDER_ACCESS*).

Internal storage. The internal data of each Android app is stored in its local storage directory */data/data/package_name/*, and Android prevents an app from accessing the directories owned by other apps (e.g., /data/data/Facebook/). In the environment of app virtualization, the app data are stored in the subdirectories of the virtualization framework (i.e., */data/data/Parallel_Space/parallel/0/package_name/*). Our penetration test sample scans the directories of the virtualization frameworks (*/data/data/framework_name/*), and attempts to access the local storage of another virtualized app (such as Facebook).

Protected external storage. Like the internal storage, the data of apps can also be stored to the system-protected external storage directory (*/sdcard/Android/data/package_name/*), and Android access control policies prevent this directory from being accessed by other apps. However, in virtualization frameworks, this directory is also redirected to a subdirectory of the framework itself. Thus, we also scan it to evaluate whether a malicious app can access the data of another app (DropBox in our experiment). Actually, DropBox uses a temporary file (e.g., *share.jpg*) in the external storage to cache the user shared files, and our penetration test sample attempts to locate and read it.

Private app component. As practice of modularity programming, many Android apps consist of multiple components that communicate with each other. Most of the components are private to the app itself, and cannot be accessed outside the app. For example, many apps use content providers as databases to store user private data. In the Firefox app, sensitive data (i.e. browser history, search history and bookmarks) are stored in a content provider *org.mozilla.gecko.db.BrowserProvider*. Once executed in a clean Android system, this component can only be accessed by Firefox itself through a specific URI (*content://org.mozilla.firefox.db.browser/*). Our penetration test sample attempts to query this content provider and steal the sensitive information.

System services. Android system services provide various sensitive operations, for example, user accounts are managed by the AccountManagerService, app downloads are managed by the DownloadManagerService. To isolate data from different apps, these services verify the app's identity before it accesses a specific sensitive resource. Our penetration test sample attempts to access the account of another app (Twitter) by querying the system service *AccountManagerService*.

Shell commands. Inherited from Linux, Android supports a set of shell commands, for example, ps and ptrace. Since malware can leverage the responses of shell commands and steal private information from other apps (e.g., side channel attacks [19]), Android enforces a set of fine-grained access controls to restrict the usage of these commands. Our experiment detects whether an app on the virtualization framework can breach the access controls and obtain log history of other virtualized apps from the shell command *logcat*, or monitor the execution environment with commands *ps* and *top*.

Socket. Android apps can utilize sockets to transfer sensitive data between different processes or communicate with remote cloud servers. Commonly, these sockets are private and inaccessible by other apps. For example, a map navigator app (BaiduMap) uses a private socket to accept commands from its voice assistance, such as starting or stopping the navigation, or changing the navigation destination. Thus, to test the protection of private sockets, we design an experiment which connects to the socket of other virtualized apps. Specifically, our penetration test sample scans and locates the sockets of a popular VPN app (*openVPN*).

3.5 Results & findings

We run our penetration test sample on 32 commercial virtualization frameworks and a clean Android 8.0 system. The following results are collected on Google Pixel. As the result shows, all the tested virtualization frameworks are vulnerable to most of the attacks. Our penetration test is successful on most of the attack targets. The only exception is *info.cloneapp.mochat.in.goast*, which provides effective defense against our attacks to system service and content provider. On the other hand, although *com.lbe.parallel.intl* and *com.parallel.space.lite* enforce some sorts of protections so that our simple attack fails to attack an unauthorized external storage, these protections can be bypassed by our relative attack and link attack. We detail the successful attacks as well as our findings as follows:

Inherit permissions from the virtualization framework. Our experiment shows that no virtualization framework checks the permission of the virtualized apps. More severely, Table 2 shows the number of permissions declared by the 10 most popular virtualization frameworks, and almost all the frameworks over-claim a large amount of permissions. Our experiment shows that an app installed in virtualization frameworks without any granted permission can easily access highly privileged system resources like device locations and SMS.

Additionally, in Android, *signature* level permissions are commonly used as higher privileged permissions than *normal* or *dangerous* ones. Such permissions are granted by the system only if the requesting app is signed with the same certificate as the app that declared the permission. For example, an app with *com.amazon.CONTENT_PROVIDER_ACCESS* permission can access user's shopping history on Amazon. Unfortunately, we observed that all *signature* level permissions are downgraded to *normal* in virtualization frameworks. Thus, malware can easily access various sensitive resources, causing severe data leakage to end users.

Arbitrarily access the internal/external storage of apps. All the 32 studied commercial frameworks do not restrict the accesses to the internal storage of apps. That is, a malicious app can easily read/write the cookies of WhatsApp, login token of Facebook, etc. Besides, many apps store their executable files (such as .so or .jar files) in their private directories (*/data/data/package_name/*). Attackers can replace these files with their own crafted ones, causing code injection attacks.

Likewise, most of the virtualization frameworks do not verify accesses to the protected external storage (i.e. */sdcard/Android/data/package_name/*). Although *com.lbe.parallel.intl* and

	#Permissions		
Package Name	AOSP	3rd Party	Total
com.lbe.parallel.intl	94	39	133
com.ludashi.dualspace	102	80	182
info.cloneapp.mochat.in.goast	163	216	379
com.parallel.space.lite	94	38	132
com.jiubang.commerce.gomultiple	102	101	203
com.excelliance.multiaccounts	102	101	203
com.ludashi.superboost	104	80	184
com.in.parallel.accounts	102	80	182
com.nox.mopen.app	97	78	175
com.polestar.domultiple	104	88	192

Table 2. The permission declarations of virtualization frameworks. The second column lists the number of Android AOSP [4] permissions declared by the frameworks. The third column lists the number of customized permissions for 3rd party vendors (e.g., Samsung and Huawei).

com.parallel.space.lite enforce access controls and can defend against our simple test, we find that they can be bypassed with the following attacks.

- **Relative Attack.** To verify an app's accessibility to a protected external file, some of the virtualization frameworks compare the package name of the requesting app with a substring of the file name. For example, suppose the external files of Facebook are stored in */sdcard/Android/data/virtualization_framework/Facebook/*, and an app Mal requests to access the file. The virtualization framework denies the access since it assumes that Mal can only access files whose name is started with */sdcard/Android/data/virtualization_framework/Mal/*. However, such mechanism accepts a relative path */sdcard/Android/-data/virtualization_framework/Mal/../Facebook/*, although it also directs to the same path as before.
- **Link Attack.** Parallel Space is the only framework which is not affected by the relative attack. However, we find it vulnerable to another kind of attack. The malware can create a link file which points to targeted external file (e.g., using command *"ln -s la /data/data/Parallel Space/parallel/0/Facebook"* to create a link to the Facebook directory in Parallel Space). Since this link file (*la*) is stored in the directory of the malware, the accessibility check passes, although it actually points to an unauthorized location.

Information leakage in system services. Revealed by our experiments, 30 of the tested frameworks lack access controls to restrict the queries to the system services. Thus, malware can steal the data of other virtualized apps. For example, Twitter stores the user credentials in a system service *AccountManagerService*. Any customized app can query this service to get all the account information stored in it, which leads to account leakage of Twitter users.

Abuse the private app components. In 30 of the tested virtualization frameworks, malware can easily access the private components of other apps. Thus, all the data in these private components are leaked to attackers. For example, Firefox uses a private content provider to store the browser histories and bookmarks of its users. Attackers can query this component using a specific URI (*Uri.parse("content://org.mozilla.firefox.db.browser/")*), which actually leaks all the sensitive user data in this content provider.

Monitor other processes through shell commands. To prevent sensitive information leakage and side-channel attacks, Android restricts the usage of shell commands (e.g., *logcat*, *ps* and *top*).

Commonly, these commands can list the runtime information of the foreground processes. For example, some apps (i.e. TP-Link) log user credentials in plain text. Utilizing these commands, existing work shows that malware can launch targeted phishing attacks [27]. Unfortunately, 29 of the 32 tested virtualization frameworks do not restrict these commands. Thus, every app in virtualization frameworks can monitor other processes by utilizing these shell commands.

Abuse socket. Android apps use private sockets to communicate with different processes or remote servers. However, 26 of the tested frameworks do not enforce effective protections to them. That is, all our tested sockets are exposed to attackers. For example, our penetration test sample can read or write data in the socket of a popular VPN app *openVPN* in virtualization frameworks. Besides, each socket domain can be bound by only one app. Thus, an attacker can preemptively bind to the socket domain of other apps and conduct DoS attacks.

3.6 Case study: several demonstrations

We conduct several exploitation demos to illustrate the real-world damage of the vulnerabilities introduced before. The demonstration videos are available on YouTube (https://youtu.be/Mk_ZISSitow), and the details are introduced below:

Abusing Facebook account. Facebook is a popular social app with numbers of users. Using its Android app, users can manage their contacts, share their photos with friends or record their daily life. To ease the usage of Facebook, it manages a token as the certificate of each login transaction. The token is generated by the Facebook server, and stored locally by the client-side app.

The login token is stored in the local directory (*/data/data/Facebook/*) of the Facebook app. Normally, thanks to the isolation mechanism of Android, an app can only access its own local storage. However, our studies show that this access control mechanism does not work on virtualization frameworks. Thus, we write a demonstration app which dumps all the local files of Facebook, and sends them to our remote server. In this server, we run an Android emulator, and pre-install the Facebook app. After the server receives the dump files from the victim, it replaces the login token of its local Facebook app with the token stolen from the victim. As a result, our remote server can login Facebook as the victim, and abuse its account (i.e. steal chatting history or send messages). The victim is unaware of the attack because Facebook allows an account to be logged in from multiple devices simultaneously.

Phishing attacks to in-app billing. In-app billing is a common feature of Android apps. Users usually bill in apps to enjoy the advanced functionalities. To the best of our knowledge, no existing phishing attack targets in-app billing, because in Android, it is hard for attackers to know when the in-app billing starts and to hijack the billing activity. However, these restrictions are not in effect within the virtualization frameworks. Our demonstration attack targets a popular wallet app (AliPay). The attack can be easily ported to other apps.

To conduct the phishing attack, our demonstration app runs a background service that monitors when the AliPay starts. Specifically, it obtains the foreground app information by calling the interface *getRunningProcessInfo()* of a system service *ActivityManagerService*. By continuously monitoring this interface, our app can be notified when the AliPay starts up. Then, we pop up a transparent phishing window to cover the top activity. This window can capture the user inputs (password for payment), and send them to our remote server.

Leaking chatting history. WeChat is a widely used chatting app in China. To protect the chatting history, WeChat encrypts these data and stores them in a private database of its internal storage. Like other local storage data, Android isolation mechanisms prevent the chatting history of WeChat from being stolen.

Due to the weakness of virtualization frameworks, our demonstration app can directly obtain the encrypted chatting history. Besides, WeChat open-sourced their encryption algorithm, which uses user account information and device information to generate an encryption key. We read the *SharedPreference* of the WeChat to get the account information of the user and query the device information from the Android system services. Then, based on the encryption algorithm, we compute the encryption key, and use it to decrypt the chatting history. Finally, we obtain all the user chatting history in plain text.

Steal/forge user email. Gmail is a popular email service developed by Google. Its Android app uses a content provider (*com.android.email.provider.EmailProvider*) to store user sensitive data such as login credential and email history. To protect this content provider, Gmail defines a *signature* level permission (com.google.androidgm.email.permission.ACCESS_PROVIDER), which only can be granted to apps signed with Google's certificate. However, our experiment shows that within virtualization frameworks, this content provider can be arbitrarily accessed by any other virtualized apps. As a result, attackers can query this content provider to obtain the email history of a Gmail user. More severely, attackers can also forge a fake email as the victim, and send it to his/her friends, conducting phishing attacks. Figure 4 illustrates a code example for forging a fake email.

```
ContentValues values = new ContentValues();
values.put("fromList", "Bob@gmail.com");
values.put("toList", "Alice@hotmail.com");
values.put("snippet", "Hello,Alice-from:Bob");
values.put("displayName", "Bob");
... ...
Uri uri = Uri.parse("content://com.google.android.gm.email.provider/message");
ContentResolver resolver = getContentResolver();
resolver.insert(uri, values);
```

Fig. 4. Code example for forging fake email in Gmail.

4 SEVERENESS OF VULNERABILITIES

To better understand how malware can attack the virtualization frameworks and virtualized apps, we systematically study the ecosystem of app virtualization. To be specific, our study proceeds from two perspectives: first, we study what kinds of apps are actually executed on virtualization frameworks by Android users. The result reveals the real-world attack surface of app virtualization. Second, we study how apps are launched and used in virtualization frameworks. This experiment shows how malware can be distributed. Finally, we provide real-world case studies and proof-of-concept exploitations to demonstrate the attack scenario in our anonymous video: https://youtu.be/Mk_ZISSitow. Our study unveils a complete exploitation scenario: a malware can be installed and launched in an app virtualization framework (§4.2), utilizes the vulnerabilities of the underlying virtualization framework (§3), and attacks the benign Android apps residing in the framework (§4.1).

4.1 Understanding the attack surface

To understand the attack surface, we first study what kinds of apps are executed on virtualization frameworks. To achieve this, we design an experiment based on the observation that user reviews illustrate the usage scenarios of apps. Thus, our experiment proceeds in three steps. First, we crawled user reviews of the top ten studied frameworks from Google Play. Despite the non-English and invalid reviews (shorter than three words), we collected 11,008 user reviews in total. Second, we crawled the names of the top 100 ranked Android apps from each category on Google Play. Then, we compare the user reviews with the app names, revealing what apps are used by users. Since users are likely to use a short abbreviation (i.e. WhatsApp) rather than the whole name of the app (WhatsApp Messenger), we apply a normalization phase as follows:

App name normalization. To normalize the app names, we first compute the occurrence frequencies for every words and phrases in app names. For example, there are two words (*WhatsApp* and *Messenger*) and one phrase (*WhatsApp Messenger*) in the name "WhatsApp Messenger". After computing the occurrence frequencies of these words and phrases in other apps' names, we find Messenger occurs much more frequently than WhatsApp. ($freq(WhatsApp)/freq(Messenger) < Threshold$). Thus, the word *Messenger* is a common word that should be excluded, and we use *WhatsApp* as the normalized app name.

App Category	#App Names	#Reviews	#Total
COMMUNICATION*	22	1,279 (39.8%)	1,824 (56.7%)
SOCIAL*	20	394 (12.3%)	
FAMILY*	13	131 (4.1%)	
DATING*	6	20 (0.6%)	
GAME	95	534 (16.6%)	534 (16.6%)
Others	108	858 (26.7%)	858 (26.7%)

Table 3. The category of apps executed in virtualization frameworks. App categories labeled with $\star$ are used for social communication. The second column lists the number of app names. The third column lists the number of corresponding reviews.

App Name	App Category	#Reviews
WhatsApp Messenger	COMMUNICATION	639
Google*	Others	171
YouTube*	Others	103
Facebook	SOCIAL	92
MESSENGER	COMMUNICATION	77
Clash of Clans	GAME	76
Gmail*	COMMUNICATION	60
WeChat	COMMUNICATION	33
Instagram	SOCIAL	22
Google Play Games	GAME	19

Table 4. Top 10 ranked apps executed on virtualization frameworks, and the number of corresponding reviews. In most virtualization frameworks, the apps labeled with $\star$ are pre-installed by default, unless the users explicitly choose not to install them.

Table 3 and Table 4 illustrate the distribution of apps executed on virtualization frameworks. In the 11,008 valid reviews, 3,216 reviews mention app names. Among them, more than 56% are related to social communication apps, such as chatting apps, social network apps or teamwork apps. We deeply investigate the corresponding reviews and find that Android users prefer to have multiple user spaces for their social network (mostly one for the private space, some others for public spaces). Social apps manage a large amount of user private data (e.g., contacts and chatting histories). Thus, leveraging the vulnerabilities above, these private information are exposed to the malware. Additionally, we notice that some users use these frameworks to manage on-line shops, or even their digital wallets (i.e., Pockets). These apps can also be targeted by malware. Another important part of apps in virtualization frameworks are game apps. Users commonly virtualize games to use game bots. We investigate a game bot community in §4.2.2.

We observed that users of the virtualization frameworks care about their privacy, and some of them even worry about the security of the virtualization frameworks [28]. However, no existing work studies the security of these frameworks systematically, thus many users still consider them secure. Indeed many users believe that virtualization frameworks use a secure sandbox to execute their apps and apps are well isolated, which makes the users ignore the threats of malware.

4.2 Understanding the malware distribution channels

To understand how malware can be distributed to the virtualization frameworks, we conduct a comprehensive study of how apps can be launched and used in these frameworks.

4.2.1 Insecure app launching. First, by manually analyzing the 32 selected frameworks, we notice that these frameworks support installing apps from various sources. Despite that all the frameworks can launch an app that is already installed to the Android device, each virtualization framework supports at least one of the following approaches to install apps:

Launch an app from the SD card. Almost all the virtualization frameworks support app installation from the SD card. To install an app, the user can first download the installation package (.apk file) to the global accessible SD card, and use the virtualization framework to load the package. In Android, content on SD card is global accessible by any app with a *READ_OR_WRITE_EXTERNAL_STORAGE* permission, thus, a malware can easily replace the installation package before it is loaded by the virtualization framework.

Embedded app market. Some virtualization frameworks embed their own app markets. However, according to our observation, most of them are vulnerable to attacks. For example, *Go Multi*, a popular virtualization framework in Google Play, uses plain-text HTTP links when users are downloading apps from its web disk servers. Thus, attackers can conduct a man-in-the-middle attack to replace the downloaded app files.

4.2.2 Insecure game app usage. By analyzing the user reviews of using virtualization frameworks for gaming, we find that many users utilize app virtualization to launch game bots. Then, we investigate a famous game bots community which uses the virtualization framework.

A game bots community using virtualization frameworks. GameGuardian is a popular game bots tool and community. Typically, it can only be executed in a rooted Android device. After March 2018, it supports being executed without rooting the device [14], which is replaced by using the virtualization frameworks. Interestingly, we observe that GameGuardian is utilizing a vulnerability we introduced to instrument the game apps. Although the vulnerability is exploited to achieve the legitimate need of GameGuardian, it is not certain that the vulnerability will not be abused.

Figure 5 depicts how game bots work with the help of app virtualization. Users should first load the GameGuardian app into the virtualization framework. Then, various game bots (written in LUA) can be loaded to GameGuardian. LUA is a powerful and lightweight programming language. Since a LUA program can easily embed posix C code [18], a LUA-based game bot can implement any of the exploitations discussed in §3.5. GameGuardian also supports a game bots market to distribute 3rd party bots. As a result, malicious developers can easily distribute their attack scripts.

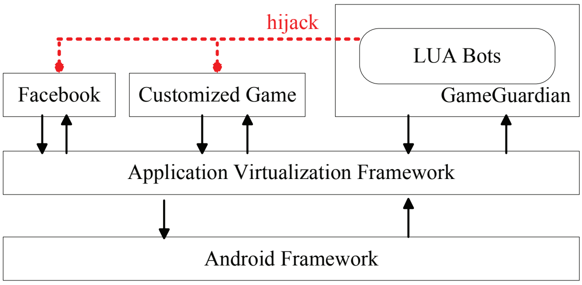

Fig. 5. The infrastructure of GameGuardian, which uses virtualization frameworks to load game bots.

4.3 A real-world demonstration

Our demonstration example (https://youtu.be/Mk_ZISSitow) illustrates a real-world attack to Facebook. In this attack, we succeed in replacing an executable file of Facebook with a crafted one, by means of a simple customized game bot.

```
 1 ParallelSpace= /data/data/com.lbe.parallel.intl/parallel/0
 2 Facebook_SO = /com.facebook.katana/lib-xzs/libcaffe2.so
 3 function replaceExecutable( path )
 4       local f2 = io.open(path,'w')
 5       f2 : write( PAY_LOAD )
 6       f2 : close()
 7 end
 8 function main()
 9       replaceExecutable( ParallelSpace + Facebook_SO )
10 end
```

Fig. 6. A LUA program that manipulates an executable file (libcaffe2.so) of Facebook.

To launch the attack, we first upload a game bot (LUA program) to the GameGuardian forum. To be ethical and avoid it been downloaded by other users, we explicitly notify that this bot is used for testing purpose. Our script passes the vetting processes, and it is visible by all the users of GameGuardian. Figure 6 shows the content of the LUA program. In this program, line 9 replaces an executable file (libcaffe2.so) of Facebook with a crafted one. Once activated, the replaced executable file takes control of the Facebook app. Then, attackers can easily steal user privacy, or abuse the user account.

5 REPACKAGE WITH APP VIRTUALIZATION

The above sections illustrate how malware can leverage the vulnerabilities of app virtualization frameworks to launch attacks. In this section, we show that malware can abuse the app virtualization techniques from another scenario: by repackaging malicious payload into Android apps using virtualization techniques. We observed that app virtualization techniques are frequently applied by developers to repackage Android apps. Repackage used to be a common approach to distribute Android malware, and a plenty of techniques are proposed to detect repackaged apps. However, they are disabled by the virtualization techniques. In this section, we first propose a new method to detect apps repackaged by virtualization techniques. Then, utilizing this tool, we drive a large scale analysis on Android apps.

5.1 Methodology

Existing approaches for detecting repackaged Android apps are based on code similarity analysis [21, 39]. Commonly, they drive static analysis to capture the signatures of apps, and calculate code similarity based on the analysis results. Figure 8.(a) illustrates an example of a repackaged app without utilizing app virtualization. In Android, the execution code of apps are packed into a file named *classes.dex*. Malware are used to directly inject the malicious payload into this file. Thus, the repackaged app and the original app share common code (the *Original App Code* in Figure 8) in their executable files (*classes.dex*). This is a common assumption of existing repackage detection approaches. However, as illustrated in Figure 8, this assumption is incorrect on repackaged apps with virtualization techniques. In this example, the original Android app is stored in the local storage of the framework (assets), and dynamically loaded at runtime. As a result, the executable file of the repackaged app is the code of the virtualization framework, thus it does not share common code with the original app.

We propose a new approach to detect repackaged apps. First, we summarize several common features of virtualization techniques, and propose a static analysis approach to detect apps with virtualization. Second, if an app virtualizes another app, we drive another analysis to determine whether the virtualization is applied to repackage the app. Our detection method is supported by two key insights. First, popular apps tend to sign their code with private signatures that malicious developers are difficult to obtain. Then, these apps commonly check their signatures before execution, preventing malicious developers from repackaging them with a different signature. Malware therefore is difficult to evade our analysis, that is, using a same signature as the repackaged app. The details of our approach are described below:

```
<activity>
  android : name =  com.lbe.doubleagent.client.proxy.ActivityProxy$P40
  android : process = :P40
  android : exported = false
  ... ...
  android : configChanges = keyboard|keyboardHidden|navigation|orientation|
      screenSize
</activity>
```

```
<activity>
  android : name =  com.lbe.doubleagent.client.proxy.ActivityProxy$P41
  android : process = :P41
  android : exported = false
  ... ...
  android : configChanges = keyboard|keyboardHidden|navigation|orientation|
      screenSize
</activity>
```

Fig. 7. A code example for similar components in virtualization apps. The code snippets are extracted from Parallel Space.

Recognizing virtualization techniques. According to our observation, the apps with virtualization techniques share the following common features: First, apps with virtualization tend to use a large number of wrapper components to provide proxies between the original app and the Android framework. These wrapper components are structurally similar to each other. For example, Figure 7 illustrates two wrapper components from Parallel Space, which have similar configurations

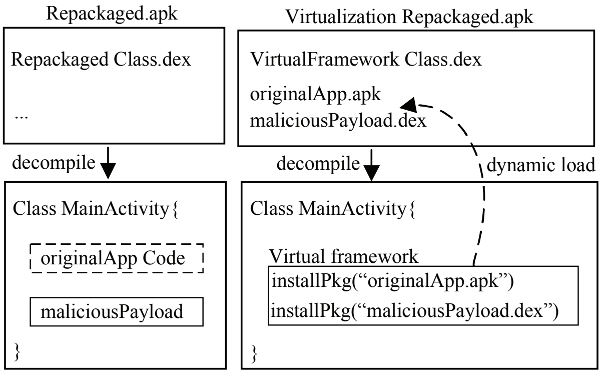

Fig. 8. Android repackaged apps with or without virtualization techniques.

(line 2 to line 6) to each other. Thus, we analyze the similarity between the components of each app. We consider two components to be similar if over 80% of their code is identical. Empirically, if more than 50 percent of the components are similar to each other, the corresponding app is very likely to use virtualization techniques. Second, apps with virtualization need to hook the original apps, thus, they commonly override the instrumentation callback (*callApplicationOnCreate()*). Our approach detects these two features, and reports an app if it matches both of them.

Identifying Repackage. An app with virtualization is not necessarily a repackaged app. A benign app can also use virtualization to achieve modularity programming. Fortunately, we observe that, different from the benign apps, a malicious repackaged app tends to repackage an app that is developed by a different party. As a result, the certificate of the original app (*originalApp.apk* in Figure 8) is different from the certificate of the repackaged app (*Virtualization_Repackaged.apk* in Figure 8). Thus, we first recognize the inlined apk files of each app. Then, we compare the signature of the inlined files with that of the root apk file, and report an app if the signatures differ. Besides, some cases encrypt their inlined apk files so that we cannot obtain their signatures. Our tool labels them as suspicious apps.

5.2 Data set

With the help of our tool, we evaluated 250,145 apps from various app stores. As depicted in Table 5, we crawled 97,304 random apps from Google Play, and 152,841 random apps from three 3rd party Android app stores.

App Market	#Apps (Total)	Download Time
Google Play	97,304	2018 April
YingYongBao	44,831	2018 April
Qihoo	104,094	2018 April
Xiaomi	3,916	2018 Spring

Table 5. Android apps collected from Google Play and 3rd party app markets.

5.3 Overall results and precision

Table 6 illustrates the overall results of our experiments. Among the evaluated apps, 180 are reported as repackaged apps with virtualization, and 29 apps are reported as suspicious apps.

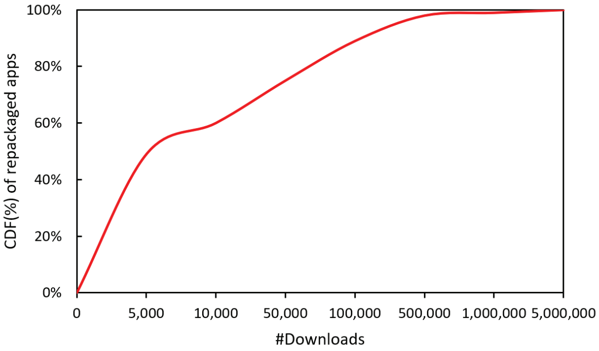

Fig. 9. The download numbers of the studied repackaged apps.

Precision. To check the false positive rate of our results, we manually verify the reported apps. Because the suspicious apps use encryption to protect their code, and it is time-consuming to decrypt them, we verify all the repackaged apps and several random samples of suspicious apps. As a result, among the 180 repackaged apps, only 16 of them are false positives. The reasons for false positives are: six apps are virtualization frameworks with 3rd party plug-in apps. Our tool mistakenly classifies them into repackaged apps because the plug-in apps have different certificates from the frameworks. The remaining 10 false positives utilize the virtualization techniques to load 3rd party apps. Our investigation shows that these apps only load a small portion of the 3rd party apps, and reuse them as a separate module. All the false positives reuse the code from the virtualized apps. However, they have no malicious intent. Furthermore, to study whether the suspicious apps can be repackaged ones, we randomly select 10 samples and attempt to decrypt them manually. As a result, we successfully decrypt four of them, in which two samples are confirmed to be repackaged apps.

	#Apps		
App Market	Using Virtualization	Suspicious	Repackaged
Google Play	52	10	6
YingYongBao	203	15	168
Qihoo	40	1	3
Xiaomi	7	3	3
Total	302	29	180

Table 6. The overall results of our experiments for detecting repackaged apps. The second column lists the total number of apps using virtualization techniques.

To study the popularity of the repackaged apps, we collect their download numbers from app markets. The results show that some of the repackaged apps earn lots of downloads. For example, one repackaged version of a popular game app *Daddy was a Thief* has more than 500,000 downloads on YingYongBao, and another repackaged app has about 390,000 downloads. Figure 9 illustrates the download numbers of these repackaged apps.

5.4 Findings

Anti-virus tools are ineffective to detect malware using virtualization techniques. To measure whether the anti-virus tools can detect malware using virtualization techniques, we upload

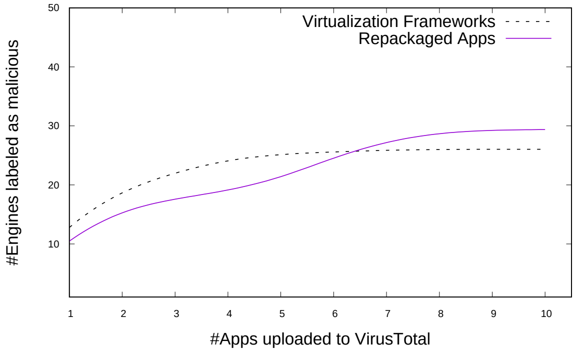

Fig. 10. The detection results of anti-virus engines on VirusTotal. The results are collected by uploading the 10 commercial virtualization frameworks and 10 randomly selected repackaged apps.

the 164 repackaged apps to VirusTotal. The results show that, none of the 64 anti-virus engines can detect all of them, and only 39 engines can detect part of the samples. We analyze the reports of these anti-virus engines, and find that 18 of them treat these apps as PUP (potential unwanted programs) or Riskware.

Furthermore, we upload 10 benign virtualization frameworks to VirusTotal, and find that they are mistakenly reported as malware by 26 anti-virus engines. Depicted by Figure 10, many anti-virus engines cannot tell apart malware from the benign apps, and show similar results to both of them.

Most of the repackaged apps are developed by a small number of groups. To study the developers of these repackaged apps, we conduct an experiment to categorize the signing certificates of these apps. As a result, we find that all these 164 true positive repackaged apps are signed with 6 certificates, which indicates that these apps are developed by a small number of developers. We manually check the code of these apps, and find that apps with a same signature have same code structures. Indeed, one of the malicious developers owns 58 repackaged apps, including *com.gamecircus.PrizeClaw*, *com.rebeltwins.aliensdrivemecrazy* and other popular apps. This indicates that virtualization techniques are capable of repackaging various Android apps, and can be helpful to generate plenty of repackaged apps.

Apps are repackaged once and no need to update. An interesting finding is that most of the repackaged apps with virtualization techniques do not update on app stores. There is only one version of each app and it has no version change logs. With a deep investigation, we find that the repackaged apps can dynamically update themselves by replacing the apk files in their virtualization modules. As a result, once a repackaged app is installed, there is no need for the app developer to update it on the app stores.

5.5 Case Study

A batch creation of repackaged apps. YingYongBao is the largest Android app market in China [5]. Our tool locates 168 repackaged apps from it. We further analyze these results and find that 58 of them are submitted by a same developer with similar package names, *com.ab.pluginX* (X stands for a random string for each app). Additionally, the original apps are placed under the *assets* directory, and renamed to *plugin.apk*. Considering the number of repackaged apps, it is highly likely that they are created from a same template.

Interestingly, among the repackaged apps, one sample embeds two apk files under its *assets* directory. One is *plugin.apk*, and the other is a mis-spelled one, *pulgin.apk*. Furthermore, the app information of this repackaged app on YingYongBao contains mixed information from these two embedded apps. Specifically, the app description is inherited from *plugin.apk*, which is a car game. However, its category is *Fly&Shoot* game, which is the same as the *pulgin.apk*. This mistake demonstrates that the repackaged apps may be created without human efforts.

A malware with severe obfuscation. Our tool locates a suspicious app, *com.bantu.hxbzyz*. According to its code in *classes.dex*, this app utilizes a virtualization framework *KXQP* to repackage an app. Using the framework, it loads two encrypted files dynamically. One of them (named com.bantu.hxbzyz.jar) is the encrypted apk file of the original app. Interestingly, the other file is hidden in a suspicious png file (*background.png*). After decryption, this png file releases another two files. One of them is a supplement package of the virtualization framework, while the other one is further encrypted. We uploaded this encrypted file to VirusTotal [37], and 16 of the anti-virus engines detect this file as a Trojan.

6 MITIGATION

In §3 and §4, we show that malware can utilize the vulnerabilities of virtualization frameworks to attack benign apps, causing privacy leakage, code injection, and other security threats. In §5, we find that benign apps are repackaged by malware with virtualization techniques. To mitigate these security threats, this section provides some recommendations to both the virtualization frameworks and the app developers.

6.1 Enhancing the security of virtualization frameworks

The security is a critical consideration of virtualization frameworks. We notice that many virtualization frameworks explicitly declare that they will not abuse permissions or leak user privacy [32, 34]. However, our experiments show that only a little effort is taken to prevent these frameworks from being abused by malware. As discussed in §2.1, virtualization frameworks act as a proxy between the virtualized apps and the Android framework, thus Android frameworks cannot control accesses of apps executed in the virtualization frameworks. To ensure the security while satisfying the virtualization demands of mobile users, OS developers can apply more flexible access controls. For example, Android provides a mechanism called “isolated process" to create isolated processes in apps. By leveraging this mechanism, app virtualization frameworks can execute virtualized apps in isolated processes, mitigating the security issues revealed in this paper. However, the isolated processes are highly restricted in Android, for example, they are forbidden to access any of the Android services, thus no app virtualization framework leverages such mechanism. It might be helpful if Android applies fine grained access controls to the isolated processes, instead of such a strong restriction to their functionality. From the perspective of these virtualization frameworks, they should inherit the role of the Android framework and re-implement the access control policies. To achieve this purpose, they should first hook the sensitive interfaces of Android, and implement the access control policies in the hook functions. Fortunately, according to our observation, almost all the frameworks already hooked all sensitive interfaces. The remaining work for them is to implement the corresponding security enforcements.

6.2 Detecting the virtualization

An app can detect whether it is being virtualized by either a virtualization framework or a repackaged app:

```
1. com.facebook.katana.LoginActivity.OnCreate()
2. android.app.Instrumentation.callApplicationOnCreate()
3. android.app.ActivitThread.handleServiceArgs()
   ... ...
4. dalvik.system.NativeStart.main()
```

(a) Call stack of Facebook in a clean Android system

```
1. com.facebook.katana.LoginActivity.OnCreate()
2. android.app.Instrumentation.callApplicationOnCreate()
3. com.doubleagent.client.e.callApplicationOnCreate()
4. com.lbe.doubleagen.client.g.a()
5. com.lbe.doubleagen.client.proxy.
                 ServiceProxyImple.onStartCommand()
6. android.app.ActivitThread.handleServiceArgs()
   ... ...
7. dalvik.system.NativeStart.main()
```

(b) Call stack of Facebook in Parallel Space

Fig. 11. Call stacks of Facebook in a clean Android system and a virtualization framework, Parallel Space.

- **Detecting instrumentation.** Virtualization techniques usually override the system instrumentation callback (*Instrumentation.callApplicationOnCreate())* of the original app. Thus, to check whether an app is executed in a virtualization framework, we can monitor the call stack and check whether the callback is replaced. For example, as illustrated in Figure 11, the original callback (line 2) in Figure 11.(a)) has been replaced by a 3rd party implementation (line 2 to line 5 in Figure 11.(b)), which illustrates that the app is executed in a virtualization framework.
- **Verifying the file system structure.** Virtualization framework redirects the file operations of the virtualized apps. Thus, to ensure whether an app is executed in a virtualization framework, one can also verify the structure of the file system. For example, the system call *getDataDir()* returns the location of the app's internal storage. In virtualization frameworks, this path is replaced by a subdirectory of the virtualization framework itself.

To prevent from being abused, apps can either alert the users that they are not executing the original app, or restrict the usage of the apps (for example, execute a “safe mode" that does not load user privacy from the server).

7 DISCUSSION

Other security impacts of the virtualization techniques. Our paper now focuses on how malware can utilize app virtualization to either attack the mobile users or to repackage Android apps. However, according to our observation, we find that not only malware, but also benign apps are using the app virtualization techniques, causing other security issues. In §5.3, our tool finds that 302 Android apps are using the virtualization techniques. Among them, 209 are repackaged or suspicious apps, and the remaining 93 apps are benign. We investigate these apps, and find that although they utilize the virtualization techniques to achieve a benign purpose (e.g., achieve modularity programming), they are all over-privileged by declaring a large number of Android permissions (as illustrated in Table 2). A deep analysis shows that most of these permissions are declared by the virtualization modules of these apps. Related researches [11, 20] studied the over-privileged problem of Android apps, and discussed the security threats imposed by this issue.

Aggressive results of anti-virus engines. Our paper illustrates that malware are armed by virtualization techniques to evade malware detection. However, since anti-virus engines cannot tell apart virtualization frameworks from malware with virtualization techniques, they aggressively mark all of them as malware. As a result, many false positives are introduced to their results. On the other hand, even such an aggressive approach is not sufficient to completely detect malware. Actually, we observed that many malware still evade most anti-virus engines. Interestingly, we observed that an open-sourced virtualization framework (VirtualApp [6]) is labeled as malware by many anti-virus engines, thus another open-source project [36] is proposed to evade all the engines.

Malware utilization of the vulnerabilities discussed in this paper. Our paper illustrates several attacks to the virtualization frameworks. We also drive a comprehensive study on the attack surface and attack scenarios. We observed an app (*com.tiqiaa.remote*) on Google Play uses the internal storage vulnerability in §3.5 to access files of another virtualized app (*com.tiqiaa.icontrol*), as illustrated in Figure 12. Though these two apps come from the same developer and use the vulnerability for a legitimate purpose, we argue that these vulnerabilities may cause severe consequences in the future. Actually, many existing malware on Android attempt to root the infected devices before they read/temper sensitive user data. Leveraging the vulnerabilities discussed in this paper, rooting the device is no longer a precondition of the attacks.

```
File file = new File("/data/data/com.lbe.parallel/parallel/0/com.tiqiaa.
    icontrol/");
if (file.exists()){
    ... ...
}
```

Fig. 12. Code snippet in a Google Play app *com.tiqiaa.remote* which attempts to access files of another virtualized app *com.tiqiaa.icontrol*.

Evasion techniques against our repackage detector. As aforementioned, utilizing the characteristics of virtualization techniques, we propose a new detection method of repackaged Android apps. Inevitably, deliberate evasion techniques can target our approach. For example, an adversary can repackage the inlined app with its own certificate. However, even if the app is repackaged, we can locate its malicious intention by calculating its similarity to popular Android apps [21, 22, 42]. Besides, malware can also obfuscate their code to evade our similarity analysis. Since application virtualization techniques create a massive number of wrapper functions, we plan to calculate the similarity of control flow structure, instead of the code, in the future. Currently, we do not witness real-world examples with the evasion techniques above. Moreover, our detector locates 164 repackaged Android apps which evade all the existing related approaches.

Comparison with other virtualization techniques. Virtualization-based techniques have been widely used in browsers and operating systems [7, 26, 40]. These virtualization frameworks commonly virtualize a complete operating system or a browser, which enforces comprehensive access controls. Malicious apps or webpages therefore should first compromise the operating system or browser before they can attack other benign apps or webpages. The app virtualization frameworks discussed in this paper is different, that is, the virtualized targets are mostly Android apps, which applies limited access controls and assumes that the underlying Android runtime environment is secure. Unfortunately, this paper shows that app virtualization frameworks break the security assumptions, making virtualized apps vulnerable to malware.

8 RELATED WORK

In this section, we review related prior research and compare our work with those studies.

Security threats from Android customizations. The app virtualization techniques are commonly applied to customize an Android app. Prior related research studied the security risks introduced by the Android system customization. Specifically, some prior works [1, 11, 12] focus on the pre-installed apps in Android factory images and report several kinds of vulnerabilities, such as over-privilege, permission re-delegation, hanging attribute references, etc. Other related studies [2, 13, 16, 38] find that customized system images modify security configurations, and the incorrect modifications bring in security vulnerabilities. These works focus on the customization of Android system images. However, this paper reveals the security issues of app virtualization, which is applied to customize apps. Thomas, et al. [35] identify that the file and memory isolation of some app virtualization frameworks may be vulnerable. Zheng, et al. [44] describe three attacks against app virtualization techniques. Different from these studies, we systematically study a more comprehensive set of vulnerabilities on a large set of app virtualization frameworks in the wild, and show a complete exploitation scenario of these vulnerabilities, by a deep analysis of the app virtualization ecosystem. Our analysis on Android app markets discovers many real-world repackaged apps that abuse the app virtualization techniques. To the best of our knowledge, it is the first systematic study of app virtualization and its security issues.

Vulnerability detection in Android. The problem of security vulnerabilities in Android has been extensively studied. Unixdomain [31] and ION [43] study the Android sockets as well as low-level heap interfaces, and report unprotected public interfaces by finding missing permission validations. Kratos [30] and AceDroid [41] reveal vulnerabilities caused by inconsistent permission enforcement in Android system. ASV [17] discovers a design trait in the concurrency control mechanism of Android system server, which may be vulnerable to DoS attacks. These works focus on vulnerabilities of Android framework. Other prior researches reveal vulnerabilities of Android apps. MalloDroid [10], Georgiev, et al. [15] and Fahl, et al. [9] focus on the unsafe usage of SSL in Android apps and other non-browser apps. Poeplau, et al. [25] study the dynamic class loading feature of Android. Luo, et al. [23] and Chin, et al. [8] reveal the vulnerabilities caused by introducing Webviews into apps without proper input verification. CryptoLint [24] shows the vulnerabilities introduced by misusing cryptographic libraries in Android apps. The above related works focus on either Android framework or apps. Our paper is different because the security threats discussed in this paper are not caused by vulnerabilities of either the Android framework or virtualized apps, but are introduced by the app virtualization techniques.

Repackage detection in Android. This paper reveals that app virtualization is applied by malware as an alternative of repackage techniques. The existing work detects app repackage based on the code similarity between apps. Specifically, PiggyApp [45] detects the repackaged apps based on the assumption that they share the same primary modules as the original apps. DroidMOSS [39] proposes a fuzzy hashing technique to locate the changes in the repackaged app. DNADroid [21] computes the code similarity with the help of program dependency graphs. AnDarwin [22] detects repackage by computing similarity based on app's semantic information. As introduced in Section 5.1, these works assume that the repackaged app and the original app share similar code in the execution file, and this assumption is not valid for repackaged apps which use the virtualization techniques. Besides, ViewDroid [42] computes the similarity between apps based on their user interface layout (various *xml*s under *layout* directory in apk). Similarly, when an app is repackaged with virtualization techniques, its layout files are different from those of the original apps.

9 CONCLUSION

In this work, we make the first attempt to systemically study the app virtualization in Android and their security threats to users. As a result, virtualization frameworks are used by more than 100 million users worldwide, and the major customization targets are the social communication apps and game apps. With a thorough study of 32 virtualization frameworks from Google Play, we propose seven attacks, and reveal that most of the frameworks are vulnerable to them. By deeply investigating their ecosystem, we show that attackers can easily distribute a malware that targets the virtualization frameworks. We present several demonstrations to illustrate these attacks. On the other side, we show that the virtualization techniques are also applied by malware as an alternative approach for repackage. To this end, we design and implement a new repackage detector, and find 180 repackaged app from four app stores. Our manual verification shows that only 16 apps are false positives. Finally, to mitigate the security threats, we propose several recommendations for both the virtualization frameworks and the app developers.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper. This work was supported in part by the National Program on Key Basic Research (NO. 2015CB358800), the National Natural Science Foundation of China (U1636204, 61602121, U1736208, 61602123, U1836213, U1836210). Yuan Zhang was supported in part by the Shanghai Sailing Program under Grant 16YF1400800. Min Yang is corresponding author of Shanghai Institute of Intelligent Electronics & Systems, and Shanghai Institute for Advanced Communication and Data Science.

REFERENCES

[1] Yousra Aafer, Nan Zhang, Zhongwen Zhang, Xiao Zhang, Kai Chen, XiaoFeng Wang, Xiaoyong Zhou, Wenliang Du, and Michael Grace. 2015. Hare Hunting in the Wild Android: A Study on the Threat of Hanging Attribute References. In *CCS*.

[2] Yousra Aafer, Xiao Zhang, and Wenliang Du. 2016. Harvesting inconsistent security configurations in custom android Roms via differential analysis. In *USENIX SECURITY*.

[3] Android. 2017. Android: 2 billion monthly active devices. https://www.youtube.com/watch?v=S_M4B-pl05M.

[4] Android. 2019. Android Open Source Project. https://source.android.com/.

[5] AppInChina. 2018. TOP 20 CHINESE ANDROID APP STORES. https://www.appinchina.co/market/.

[6] asLody. 2018. VirtualApp. https://github.com/asLody/VirtualApp/tree/master.

[7] Bromium. 2019. Browser Isolation with Microsoft Windows Defender Application Guard (WDAG): What It Does, How It Works and What It Means. https://www.bromium.com/browser-isolation-with-microsoft-windows-defender-application-guard/.

[8] Chin Erika and Wagner David. 2013. Bifocals: Analyzing WebView Vulnerabilities in Android Applications. In *WISA*.

[9] Sascha Fahl, Marian Harbach, and Perl Henning. 2013. Rethinking ssl development in an appified world. In *CCS*.

[10] Sascha Fahl, Marian Harbach, and Thomas Muders. 2012. Why eve and mallory love android: an analysis of android ssl (in)security. In *CCS*.

[11] Adrienne-Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android permissions demystified. In *CCS*.

[12] Adrienne-Porter Felt, Helen J. Wang, and Alexander Moshchuk. 2011. Permission Re-Delegation: Attacks and Defenses. In *USENIX SECURITY*.

[13] Roberto Gallo, Patricia Hongo, and Ricardo Dahab. 2015. Security and system architecture: Comparison of android customizations.. In *WISEC*.

[14] GameGuardian. 2018. No root via Parallel Space Lite on x86 - GameGuardian. https://gameguardian.net/forum/gallery/image/447-no-root-via-parallel-space-lite-on-x86-gameguardian/.

[15] Martin Georgiev, Subodh Iyengar, and Suman Jana. 2012. The most dangerous code in the world: validating ssl certificates in non-browser software. In *CCS*.

[16] Michael Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. 2012. Systematic detection of capability leaks in stock Android smartphones.. In *NDSS*.

[17] Heqing Huang, Sencun Zhu, Kai Chen, and Peng Liu. 2015. From system services freezing to system server shutdown in android: All you need is a loop in an app. In *CCS*.

[18] Programming in LUA. 2018. An Overview of the C API. https://www.lua.org/pil/24.html.

[19] Infosec institute. 2018. Exploiting Unintended Data Leakage (Side Channel Data Leakage). http://resources.infosecinstitute.com/android-hacking-security-part-4-exploiting-unintended-data-leakage-side-channel-data-leakage/#gref.

[20] Jeon Jinseong, Micinski Kristopher K., and Vaughan Jeffrey A. 2012. Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications. In *SPSM*.

[21] Crussell Jonathan, Gibler Clint, and Chen Hao. 2012. Attack of the Clones: Detecting Cloned Applications on Android Markets. In *ESORICS*.

[22] Crussell Jonathan, Gibler Clint, and Chen Hao. 2013. AnDarwin: Scalable Detection of Semantically Similar Android Applications. In *ESORICS*.

[23] Tongbo Luo, Hao Hao, and Wenliang Du. 2011. Attacks on WebView in the Android system. In *ACSAC*.

[24] Egele Manuel, Brumley David, Fratantonio Yanick, and Kruegel Christopher. 2013. An empirical study of cryptographic misuse in android applications.. In *CCS*.

[25] Sebastian Poeplau, Yanick Fratantonio, and Antonio Bianchi. 2014. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *NDSS*.

[26] Qemu. 2019. QEMU, the FAST! processor emulator. https://www.qemu.org.

[27] Yinfeng Qiu. 2012. Bypassing Android Permissions: What You Need to Know. https://blog.trendmicro.com/trendlabs-security-intelligence/bypassing-android-permissions-what-you-need-to-know/.

[28] Quora. 2016. Is the app parallel space on my android phone safe to use is there no risk of hacking or anything like that? https://www.quora.com/Is-the-app-parallel-space-on-my-android-phone-safe-to-use-is-there-no-risk-of-hacking-or-anything-like-that.

[29] Quora. 2016. What is the process of creating bots for Android games? https://www.quora.com/What-is-the-process-of-creating-bots-for-Android-games.

[30] Yuru Shao, Jason Ott, Qi Alfred Chen, Zhiyun Qian, and Z Morley Mao. 2016. Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework. In *NDSS*.

[31] Yuru Shao, Jason Ott, Yunhan-Jack Jia, Zhiyun Qian, and Z.Morley Mao. 2016. The Misuse of Android Unix Domain Sockets and Security Implications. In *CCS*.

[32] Excelliance Tech. 2018. Multiple Accounts:Parallel App. https://play.google.com/store/apps/details?id=com.excelliance.multiaccounts.

[33] LBE Tech. 2018. Over 100 million users worldwide. https://www.facebook.com/parallelspaceapp.

[34] LBE Tech. 2018. Parallel Space - Multiple accounts & Two face. https://play.google.com/store/apps/details?id=com.lbe.parallel.intl.

[35] Julien Thomas. 2018. In-App virtualization to bypass Android security mechanisms of unrooted devices. https://2018.bsidesbud.com/wp-content/uploads/2018/03/julien_thomas.pdf.

[36] tiann. 2018. fuck_anti_virus.gradle. https://gist.github.com/tiann/42f829ae86b90934c8467f6f76dd6a85.

[37] VirtusTotal. 2018. VirtusTotal. https://www.virustotal.com.

[38] Lei Wu, Michael Grace, Yajin Zhou, Chiachih Wu, and Xuxian Jiang. 2013. The impact of vendor customizations on android security. In *CCS*.

[39] Zhou Wu, Zhou Yajin, and Jiang Xuxian. 2012. Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces. In *CODASPY*.

[40] Xen. 2019. Xen project. https://www.xenproject.org.

[41] Aafer Yousra, Huang Jianjun, and Sun Yi. 2018. AceDroid: Normalizing Diverse Android Access Control Checks for Inconsistency Detection. In *NDSS*.

[42] Fangfang Zhang, Heqing Huang, and Sencun Zhu. 2014. ViewDroid: Towards obfuscation-resilient mobile application repackaging detection. In *WISEC*.

[43] Hang Zhang, Dongdong She, and Zhiyun Qian. 2016. Android ION Hazard: The Curse of Customizable Memory Management System. In *CCS*.

[44] Cong Zheng, Tongbo Luo, Zhi Xu, Wenjun Hu, and Xin Ouyang. 2018. Android Plugin Becomes a Catastrophe to Android Ecosystem. In *RESEC*. ACM.

[45] Wu Zhou, Yajin Zhou, and Michael Grace. 2013. Fast, scalable detection of piggybacked mobile applications. In *CODASPY*.

Received November 2018; revised December 2018; accepted January 2019