

# Università degli Studi di Padova

---

DEPARTMENT OF MATHEMATICS *TULLIO LEVI-CIVITA*

MASTER'S DEGREE IN CYBERSECURITY

## **Towards Secure Virtual Apps: Bringing Application-Level Isolation to Android Virtualization**

SUPERVISOR

Prof. Losiouk Eleonora  
University of Padua

CANDIDATE

Boscolo Meneguolo Luca  
2113488

---

ACADEMIC YEAR 2024/2025

JULY 25, 2025



# **ACKNOWLEDGMENT**

I would like to express my sincere gratitude to Prof. Eleonora Losiouk for her valuable supervision and continuous support throughout the development of this thesis.

I am also grateful to those who, directly or indirectly, supported me throughout this journey. In particular, my family and friends, whose encouragement made a real difference during the more challenging moments.

# ABSTRACT

App-level virtualization is a technique that allows to execute apps in a virtual environment, without them being physically installed on the device. In the context of Android, VirtualApp is among the most known virtualization frameworks, as it gained popularity on the community mainly for its ability to set multiple social media accounts on the same device.

VirtualApp, due to technical limitations and the lack of a proper secure architecture, proved to break the Android Sandbox model, which is among the security mechanisms that grant app isolation. This vulnerability might be exploited by a malicious entity to get the user's personal data without consent.

The purpose of this work is to better understand which attacks can be performed on VirtualApp, develop a strategy to prevent them, and implement a working proof of concept on the framework. An evaluation of real-world virtualization frameworks downloaded from the Google Play Store reveals that some of them share the same security vulnerability, exposing millions of users. A final discussion stresses the importance of secure software development practices.

# INDEX

<b>ACKNOWLEDGMENT</b>	<b>III</b>
<b>ABSTRACT</b>	<b>IV</b>
<b>1 INTRODUCTION</b>	<b>1</b>
<b>2 BACKGROUND</b>	<b>3</b>
2.1 Package Name . . . . .	3
2.2 Repackaging . . . . .	4
2.3 Multi-User Functionality . . . . .	5
2.4 Virtualization . . . . .	7
2.5 App-Level Virtualization – VirtualApp . . . . .	7
2.5.1 VirtualXposed . . . . .	8
VirtualCore . . . . .	10
Proxying of Android Services . . . . .	10
File System Virtualization . . . . .	10
APK Loading . . . . .	11
2.5.2 Vulnerabilities . . . . .	11
Permissions . . . . .	11
Sandbox . . . . .	12
<b>3 RELATED WORK</b>	<b>13</b>
<b>4 THE ANDROID SANDBOX</b>	<b>15</b>
4.1 Scoped Storage . . . . .	15
4.1.1 Internal Storage . . . . .	16
4.1.2 External Storage . . . . .	17
4.1.3 Installation Directory . . . . .	17
4.1.4 System Directories . . . . .	18
4.2 Make use of the Scoped Storage . . . . .	19
4.2.1 Direct File Access . . . . .	19
4.2.2 MediaStore API . . . . .	19
4.2.3 Storage Access Framework . . . . .	21

4.3	Sandbox Vulnerabilities in VirtualApp . . . . .	22
4.3.1	Direct File Access - VirtualApp . . . . .	23
4.3.2	MediaStore - VirtualApp . . . . .	25
4.3.3	Storage Access Framework - VirtualApp . . . . .	26
<b>5</b>	<b>SECURING DIRECT FILE ACCESS</b>	<b>27</b>
5.1	General Overview . . . . .	27
5.2	Key Components . . . . .	31
5.2.1	Path Tokenizer . . . . .	31
5.2.2	PathChecker . . . . .	32
5.2.3	LibCoreStub . . . . .	35
<b>6</b>	<b>SECURING MEDIASTORE</b>	<b>37</b>
6.1	General Overview . . . . .	37
6.2	Key Components . . . . .	41
6.2.1	MediaStore File Parser . . . . .	41
6.2.2	MediaStore Cache . . . . .	42
6.2.3	Virtual Media Provider . . . . .	42
6.2.4	Media Provider Service . . . . .	44
6.2.5	Media Provider hook . . . . .	45
6.2.6	MediaStore Entry . . . . .	51
6.2.7	MediaStore Cursor Parser . . . . .	52
6.3	Conflict and Fix . . . . .	54
6.3.1	Invocation Stub Manager . . . . .	55
6.3.2	Locked Operation . . . . .	56
<b>7</b>	<b>EVALUATION</b>	<b>58</b>
7.1	Phase One – Test on Malicious Apps . . . . .	59
7.1.1	Direct File Access . . . . .	59
7.1.2	MediaStore . . . . .	61
7.2	Phase Two – Test on Legitimate Use Case . . . . .	62
7.3	Phase Three – Check for Vulnerabilities . . . . .	64
7.3.1	Time Cost . . . . .	64
	Direct File Access . . . . .	64
	MediaStore . . . . .	67
7.3.2	Fast MediaStore Operations . . . . .	69
7.3.3	MethodProxy Temporary Disabling . . . . .	70
<b>8</b>	<b>CONCLUSION</b>	<b>72</b>
8.1	Secure by Design . . . . .	72
8.2	Real World Frameworks Tested . . . . .	73

## INDEX

---

<b>9 FUTURE WORK</b>	<b>75</b>
9.1 VirtualPatch . . . . .	75
9.2 Full Android Virtualization . . . . .	76
<b>REFERENCES</b>	<b>77</b>

# 1

## INTRODUCTION

On modern smartphones, users can typically install only a single instance of a given app. While this is generally not an issue for utility apps, it becomes limiting for apps that require user accounts. Many users own multiple accounts for the same service, for example a personal and a business social media account. The inability to run multiple instances of the same app restricts their ability to manage these accounts conveniently.

One way to address this need comes directly from the developers. Some social media apps are indeed being updated to include the multi-account functionality, where one or more accounts can be added and managed. This way, users can have a seamless experience of the service in all of their managed accounts.

However, some developers are still reluctant to adopt this feature. A notable example is WhatsApp, where multiple accounts were released for Android as an update only in late 2023 [1], while iOS users are still waiting for this feature to pass the beta stage [2].

Obviously, there is still a large number of apps that do not offer this feature.

In these years, developers tried to find solutions to allow users to run multiple instances of the same app on their smartphone. App-level virtualization, being the most successful method in Android, gained notability since 2015 as it allows executing apps in isolated environments, without them being physically installed on the device. In the context of Android, VirtualApp [3] is among the first and most known virtualization frameworks.

Users may not be aware though, that app-level virtualization might break some of the key security mechanisms featured in Android. Research conducted on the virtualization framework Parallel Space by Dai et al. [4] showed that app-level virtualization exposes user to attacks, as *apps are not completely isolated from each other*.

## 1. INTRODUCTION

---

A recent work by Alberto Lazari [5] showed that VirtualApp breaks the Android permission model. This vulnerability leads virtual apps to implicitly grant access to any permission defined on the Manifest, without user intervention. In his work, he showed an attack sample and proposed a proof of concept solution for addressing this vulnerability.

A theoretical analysis on VirtualApp arose the possibility that it could also break the Android sandbox. The Android sandbox is a security mechanism that isolates each app on the device on its own separate environment. Each app has access to its own files, data and memory, and communication is only permitted with safe means. In short, the Android sandbox ensures that apps operate independently and securely, reducing the risk of data leaks and unauthorized access.

In this work are shown the sandbox-related vulnerabilities that affect VirtualApp, and a proof of concept of the implementation to address them.

Finally, a test on real world and modern app-level virtualization frameworks is performed, with the objective of determining whether the same threat affects also finished products.

# 2

## BACKGROUND

### 2.1 PACKAGE NAME

In Android, the package name is a string defined in the `AndroidManifest.xml` file, and its purpose is to uniquely identify an app.

```
<manifest
  xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.example.myapp">

  <application

    <!-- Activities and components go here -->

  </application>

</manifest>
```

Listing 1 — `AndroidManifest.xml` file showing the package declaration.

The package name, showcased in Listing 1, also serves a fundamental role in defining permissions, intents, and component names like activities and services. Each action performed by a component is associated with the app's package name, ensuring proper routing and security within the Android system.

On an Android device, only one app with a given package name can be installed at a time. In other words, any app in a device must have a unique package name. When the

user tries to install an APK file whose declared package name is the same of a currently installed app, the system will try to update the app. Because of this, running multiple instances of the same APK file is not possible.

It should be noted that the package name does not guarantee the legitimacy of an APK file. The legitimacy of the APK is verified at update time by comparing its signature with the previously installed APK. This process is done automatically by the Android package installer: if an app is originally installed by a trusted source (e.g. Google Play Store), any APK with matching package will fail to install if signatures do not match. If the app is originally installed from untrusted sources though, Android will not provide any warning.

## 2.2 REPACKAGING

A first solution for running more instances of the same app is by repackaging the desired APK with a different package name. The technique requires modifying the package name to any string that is not already shared by a previously installed APK. When installing the two apps, original and modified, Android will treat them as totally different programs.

This technique is however very impractical. Source code is needed in order to create a new build with a different package name. If the source code is not available, which is a very common situation since most apps are not open source, a proper reverse engineering process is required. The APK must be decompiled with programs like Apktool [6], the package must then be changed to a non-matching one, and the APK must be signed back again. The signature will not be the official one from the original developer, but this is not a problem because Android does not verify the legitimacy of a signature *a priori*.

This procedure is not feasible for casual users, as deep reverse engineering skills are required. It should be noted that repackaging APK files will almost surely break the terms and conditions of the author, and may even be illegal in some jurisdictions, especially if the source code is closed or patented.

It should also be noted that installing repackaged APK files from external and untrusted sources is very risky, since they might also include malicious modifications on the app code. A notable example is the family of malware identified by the name of Shedun [7], that appeared in late 2015. Hackers would target apps like Twitter, Facebook, WhatsApp and other popular services, by repackaging them with adware [8]. It is estimated that

ten million devices were infected by this malware in any of the existing variants, making it one of the most widespread Android malware at the time.

### 2.3 MULTI-USER FUNCTIONALITY

Another technique to run multiple instances of the same app revolves around the core nature of Android: the Linux kernel. Linux is a multi-user kernel, a property that is then inherited by Android as well. Each Android user has its own set of installed apps, settings and files. By exploiting this feature, it is possible to have two or more instances of the same app installed on the same device, and set different accounts to them.

A distinction between Android and Linux should be made. From a technical point of view, Android apps are mapped to Linux UIDs. This means that each app is a Linux user. Android users, as shown in Table 1, are managed by ranges on the UID.

UID Range	Purpose
0	root
10000 – 19999	Apps for <b>User 0</b>
20000 – 29999	Apps for <b>User 1</b>
30000 – 39999	Apps for <b>User 2</b>
...	...

Table 1 – How UIDs are mapped in Android.

This solution however, is far from perfect. The multi-user management is similar to the ones seen in desktop PCs, where there can only be one active per session. This means that the operating system can access only to the apps and files that belong to the current active user. Let's assume we have two users, 0 and 1. While user 0 is logged, only apps and resources related to user 0 are loaded. This in turn results to poor functionality of the duplicated apps. Data created by one user needs extra care to be shared across users. Most importantly, apps from user 1 will not run when user 0 is active, not even in the background.

The final experience is that apps will not send notifications, and cannot run periodic background tasks. In general this experience may be enough in some circumstances, but is definitely unacceptable for messaging and social media apps, where timeliness of the notifications is among the crucial factors for a proper user experience.

## 2. BACKGROUND

---

One other notable downside of this technique has to do with vendors. The Android Open Source Project (AOSP) is the vanilla flavor of the operating system, and is installed on Google Pixel devices along with Google's proprietary GApps. AOSP has had the support for multiple users since Android 4.2, however not every device is compliant with this feature.

Some vendors actually disable the multi-user feature in the customized versions of Android they install on their smartphones. This is usually justified by the fact that smartphones are personal and generally used by only one user. An example is Samsung's OneUi, where the multi-user support is long gone and missing since more than three major revisions. There are not official statements about this choice, but it seems that the most notable reason is the aforementioned, since the software that is installed on Samsung tablet devices has the multi-user support. Samsung is not the only vendor to cut the support for this feature. Chinese vendors usually include hard customizations to the official AOSP, including radical changes to the firmware and kernel.

The following is a non-exhaustive list of Android ROMs that do not officially support multi-user:

- OneUi, by Samsung
- MIUI, by Xiaomi/POCO
- RealmeUi, by Realme
- ColorOS, by OPPO
- Funtouch OS, by Vivo
- OxygenOS, by OnePlus

Some have unofficial support by workarounds, but the results might vary. In general, ROMs that are the closest to AOSP still have the multi-user feature enabled.

Follows a non-exhaustive list of ROMs with support for multi-users:

- Stock Android, on Google Pixel devices
- My UX, by Motorola
- Xperia UI, by Sony
- Fairphone OS, by FairPhone

### 2.4 VIRTUALIZATION

The most promising technique to run multiple instances of the same app is through virtualization.

Full Android virtualization, like running the entire operating system in a virtual machine, is mostly not supported. Virtualization requires dedicated hardware support, which is not available on all Android devices. Moreover, apps are sandboxed, which means that they do not share resources. Full OS virtualization would allow bypassing Android's app sandbox, which breaks the core security model. In the recent Android 16 update, Google added support for full virtualization of a Linux VM, though this feature is currently supported only by Google Pixel devices. This feature is obtained thanks to the *Android Virtualization Framework*, and allows for running a full Debian-based Linux distribution, with hardware acceleration support and a display server [9].

The Android Virtualization Framework [10] is an official platform designed to support the virtualization of Android apps, with emphasis on security. It allows developers to create virtualized Android environments that are separate from the system. It's not necessarily focused on running a completely separate virtual Android app, but instead on virtualizing specific tasks for enhanced security. AVF is not designed to allow arbitrary apps to be virtualized by end users. Its primary use case is for creating secure, isolated workloads under system control. Because of this, AVF is not suitable for running multiple instances.

### 2.5 APP-LEVEL VIRTUALIZATION – VIRTUALAPP

While full Android virtualization is not permitted, projects on *application-level virtualization* have been carried with success since 2015. One of the most known virtualization framework is *VirtualApp*, which allows running virtualized instances of apps by making clever use of the Java reflection library. VirtualApp, developed by Lody et al. [3], is an open-source<sup>1</sup> app-level virtualization framework that allows for virtual execution of APKs, without the need of installing them on the system.

---

<sup>1</sup>The source code on the official repository is updated only up to December 2017, and the license strictly forbids any business profit. Academic research is permitted.

## 2. BACKGROUND

---

In this thesis, the following terms will be used with specific meanings, which are defined below to ensure clarity and consistency:

- **Framework:** the set of software components that manage app virtualization. It includes the necessary logic to simulate a complete Android environment, intercept system calls and grants an intermediary layer between plugin apps and container app.
- **Container:** the installed application that hosts VirtualApp’s framework. It is seen as a unique app, though it runs instances of many plugin apps.
- **Plugin:** the actual target APK to be virtualized.

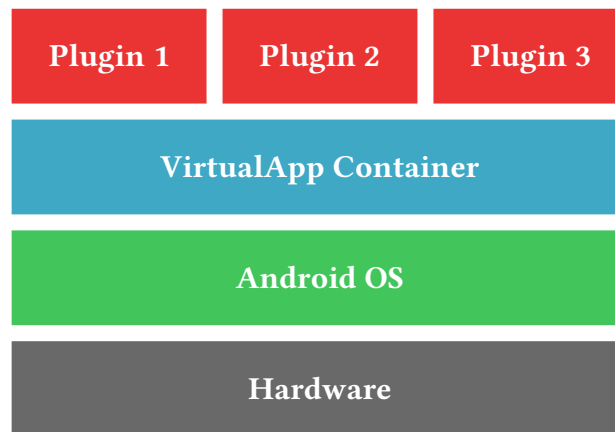


Table 2 – Depiction of the layers in app-level virtualization. The container app is installed on the system, and acts as a bridge between OS and plugin apps.

### 2.5.1 VIRTUALXPOSED

The work by Alberto Lazari includes a series of improvements over the base VirtualApp. His work builds upon *VirtualXposed* [11], which is itself a fork of VirtualApp.

The VirtualApp repository only provides the core virtualization framework, which is intended to be embedded within another application. The developers of VirtualXposed extended the VirtualApp project by embedding a launcher, allowing users to install and start apps directly from a convenient and familiar UI. Moreover, they added support for Xposed.

Xposed is a deprecated framework, used to hook methods and modify apps at runtime. It used to be popular during Android 4.4 era, when modules were developed to modify system behavior. The greatest caveat for running Xposed modules—root access—requires technical skills and usually voids the device warranty. VirtualXposed allows for running Xposed modules on the virtual environment without root, making this project

## 2. BACKGROUND

---

appealing for users that want to have a certain degree of freedom but do not want to root their devices.

While this feature offered flexibility, in the context of Alberto's research the Xposed functionality was not used. VirtualXposed was chosen primarily for the convenience of its embedded launcher, and the base support for Android 12.

Alberto improved VirtualXposed by adding unofficial support up to Android 15. Then, the virtual permission model was implemented, to address the permissioning vulnerability.

My research directly continues his work.

VirtualXposed uses a hooking framework called *Epic* [12]. Epic is actually more than a hooking framework, as the documentation reports as typical use cases aspect-oriented programming, instrumentation, sensitive API check, smash shell etc.

The working of Epic, like many other hooking mechanisms, relies on `ArtMethod` [13], [14], [15]. `ArtMethod` is a low-level data structure that is used to link Java classes in `.dex` files to native compiled code. That is because, when transitioning from Dalvik to ART runtime, a new *ahead of time* compilation was introduced. With this option, some methods are flagged as *hot*, meaning that they have a greater probability of being called. To improve performance, these methods are already compiled to native code.

`ArtMethod` holds a representation of each Java method, and a pointer to the native code. Hooking is performed by substituting this pointer's value with the one of the method that is defined by the developer. This in turn allows for the hooked function to actually execute the custom defined method.

Epic is used as a bridge to apply Xposed modules. In short, Epic is the back-end hooking framework that actually hooks the methods. Xposed is the front-end, mainly for the high demand and choice of modules. Each Xposed module is parsed, and all hooks are converted to Epic hooks, allowing for a seamless experience.

There are some limitations though. First, Epic is not maintained anymore, and the latest official release supports only Android up to version 11. Also, since `ArtMethod` deals with compiled code, the entire framework is architecture-dependant. In the context of Epic, only thumb2 and arm64 architecture are supported.

Follows a basic introduction of the key features of the framework.

### **VIRTUALCORE**

VirtualCore is the core component for the VirtualApp framework.

Its main purposes are the following:

1. **App virtualization management**

Allows for plugin app management. Plugin apps are loaded and parsed. The framework also allows for managing virtual processes.

2. **Process isolation and redirection**

Controls and launches plugin apps. Plugin apps run in dedicated processes that are spawned by the framework. It also re-routes component calls through the virtual environment, so that it seamlessly blends with Android.

3. **System Service Proxying**

At startup, VirtualCore hooks Android system services (like `ActivityManager`, `PackageManager`, etc.) using reflection and proxies. It also redirects system calls made by plugin apps, in a way for them to be ultimately called by the container app.

4. **Environment Setup**

Sets up the file system and app data directories for plugin apps to work properly. Controls the virtual app's view of the system (e.g. fake installed packages, fake user info).

5. **User Space Management**

Supports multi-user profiles within the virtual space. This way, it is possible to have multiple clones of the same app, even within the framework.

### **PROXYING OF ANDROID SERVICES**

VirtualApp makes use of Java `Proxy` instances to capture method calls. Plugin apps run in a virtualized environment, and all calls must be redirected to the system through the container app. When the virtual environment is started, a series of Android System Services is proxied using the Java reflection library. These services include critical components such as the `ActivityManager`, `PackageManager`, `ClipboardManager`, and others that are essential for app lifecycle management, package resolution, and inter-process communication. By creating `java.lang.reflect.Proxy` instances for these services, VirtualApp can intercept method calls made by the plugin app and reroute them through custom handlers defined within the container. The handlers may modify or alter some parameters, or implement a custom logic that ensures the correct redirection of system components.

### **FILE SYSTEM VIRTUALIZATION**

VirtualApp creates a virtual file system on its internal storage, showed in detail in subsection 4.3.1, that mimics the original Android directories. The framework also performs a redirection to the virtualized file system.

## 2. BACKGROUND

---

Let's assume a plugin app wants to access the internal storage. To do so, it would execute code similar to that shown in Listing 2.

```
File myFile = new File(getFilesDir(), "myfile.txt");
```

Listing 2 – Basic file access.

In a virtualized environment, the plugin app is not physically installed, and thus does not have a private internal storage folder. If the redirection was not in place, `getFilesDir()` would return the path for the internal storage of `VirtualApp`, which would be shared across all plugin apps. Each plugin app is assigned a virtual internal storage, and redirection allows for plugin apps to work as intended.

### APK LOADING

`VirtualApp` does not need the plugin apps to be fully installed on the device. The APK is copied on the virtual file system, in the respective virtual installation folder. The APK's Manifest is then parsed, and information about the package and the UID are stored. `VirtualApp` manages a mapping between real UID and virtual UID. All system calls (e.g. file access, package queries) are redirected to reflect the virtual UID space.

### 2.5.2 VULNERABILITIES

Unfortunately, as is common in the IT industry, `VirtualApp` adheres to the principle of prioritizing functionality over security. Such principle refers to a design or development approach where delivering features and usability is prioritized over implementing a strong security model. By performing app-level virtualization, we are in fact exposing ourselves to potential security threats.

### PERMISSIONS

The Android permission model allows apps to request access to sensitive resources (like camera, location, contacts, etc.) in a controlled and user-consent-based manner, ensuring user privacy and system security.

`VirtualApp` must ensure that every plugin app works as expected. To do so, it defines all the existing permissions on its Manifest. When the user installs the virtualization framework and first launches the app, a list of permission-related dialogs pops up. It is then required for the user to grant access to all the permissions, as denying one will surely cause issues when running a plugin app that requires it.

## 2. BACKGROUND

---

This solution surely works, but exposes the user to severe vulnerabilities. First, the user cannot have fine-grained control over the permissions. By adding an app to the framework and executing it, the user implicitly grants all the permissions that the virtual app requires, as those are inherited from the container. Such vulnerability is even more dangerous when combined with overpermissioning, which is unfortunately a bad coding practice among many software developers, where unused permissions are left on the Manifest file.

Studies on this vulnerability have been carried by Alberto Lazari [5].

### **SANDBOX**

A security implication of app-level virtualization comes from the fact that all plugin apps run in a virtualized environment under the container itself. This leads to the bypassing of the Android sandbox model. Virtual apps can in fact mutually access to their respective internal storage, leading to sensitive data leakage.

When dealing with official and trusted apps, we should expect them to not leverage such vulnerabilities. But with the high demand of app-level virtualization, we should consider that there is monetary interest on leveraging such vulnerabilities in malicious apps that steal personal information.

# 3

## RELATED WORK

Concerns regarding application-level virtualization are well documented in recent literature.

The research conducted by Dai et al. [4] on virtualization frameworks state that *apps are not completely isolated from each other*. Another notable paper by Zhang et al. [16] shows a comprehensive analysis of how Android app-virtualization systems create security risks. Work by Luo [17] shows a proposal that demystifies the Android plugin technology in depth, explains the underlying attack vector and investigates fundamental security problems, and proposes a lightweight defense mechanism and release a library, named PluginKiller. Such library prevents an Android app from being launched by the host app using the Android Plugin technology.

Some researchers discovered severe attacks. Research conducted by Alecci et al. [18] shows an attack to fully bypass existing detection defenses. It works by disguising itself as a benign add-on app. After installation, it spawns a controlled virtual environment that loads and runs the targeted app alongside the attacker's code to harvest sensitive data.

Some of the research stress how widespread are repackaging attacks. A work by Zerbini et al. [19] presented a mechanism to detect the intrinsic features of the virtualization technique, to avoid repackaging attacks. A work by Zheng et al. [20] shows that virtualization can be exploited for repackaging attacks to subvert app security. A work by Ruggia et al. [21] presents a novel framework that protects Android apps by embedding them in a secure, runtime-monitored virtual container. This effectively prevents attackers from repackaging or virtualizing apps to insert any malicious behavior.

### 3. RELATED WORK

---

Other sources, like a work by Shi et al. [22] addresses a rising threat where Android malware authors hide malicious behavior inside containers that leverage virtualization, rather than directly repackaging benign apps.

# 4

## THE ANDROID SANDBOX

Sandboxing is among the most impactful features in the Android security model. Every application is executed in an isolated environment, called *sandbox*, that ensures that data and resources are not accessible to other apps, unless they are explicitly shared. The Linux kernel comes into play, as every app is assigned to a unique user UID. The UID is used at lower levels to apply restrictions on file access and other system resources.

In short, sandboxing offers:

- **Data integrity:** an app cannot modify another app's files.
- **Privacy:** an app cannot read another app's data.
- **Containment:** a compromised app cannot propagate the attack.

Before analyzing the specific vulnerabilities of VirtualApp's sandbox implementation, it is essential to first understand how storage is structured and managed in Android 14.

### 4.1 SCOPED STORAGE

*Scoped storage* is the current model Android uses to manage files and directories. It was introduced in Android 10, but developers could opt out by specifying on the manifest file the line shown in Listing 3.

```
requestLegacyExternalStorage="true"
```

Listing 3 – Opt out from scoped storage. On newer versions of Android, it is not anymore possible.

Starting from Android 11, developers could no longer opt out, effectively mandating scoped storage for all apps.

The main purpose of Scoped Storage is to limit app access to external storage, granting the privacy of app and user data.

### 4.1.1 INTERNAL STORAGE

Internal storage refers to the directories that are privately accessible only to the owner package name. The owner app has complete read and write access to its internal storage, while no other app is permitted to read from or write to it.

The internal storage is located on the `data` directory, returned at runtime by calling `context.getDataDir()`.

```
└── data/user/0/com.example.packageName/  
    ├── cache  
    ├── code_cache  
    ├── files  
    └── shared_prefs
```

Figure 1 – Internal storage relative to package name `com.example.packageName`.

The internal storage directory, `data/user/0/com.example.packageName` on the example in Figure 1, contains all app-specific files, and is completely wiped when the respective app is uninstalled from the system. Common use case of the internal storage include:

- Caching app-specific files;
- Saving generic data;
- Storing user preferences. The preferences created and managed using the `sharedPreferences` API are stored in an XML format under the `shared_prefs` folder.

### 4.1.2 EXTERNAL STORAGE

External storage refers to the locations that are shared across apps. Files stored under the external storage are thus public, and are not protected by strong access control policies. For this reason, the Android documentation explicitly recommends using internal storage for storing user-sensitive data.

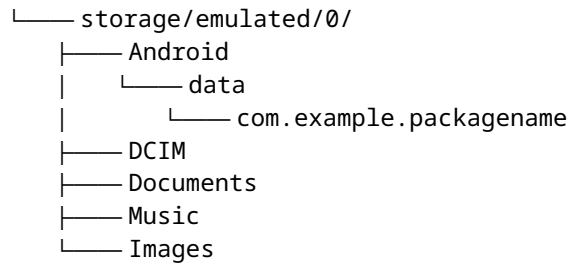


Figure 2 — External storage.

External storage, as shown in Figure 2, is located on the `storage/emulated/0/Android/data/com.example.packageName/` directory. Directories here are organized by package name, and each app has easy access to its external storage by calling `context.getExternalFilesDir()`. Scoped storage does not require the owner app to have access to additional permissions. An app can in fact access its own external storage freely. However, external storage is shared, meaning that by declaring `WRITE_EXTERNAL_STORAGE` permission, an app is allowed to access other app's external storage. These directories defined by the package name are also deleted when the owner app is uninstalled, just like internal storage.

Directories located on `storage/emulated/0/` are also considered external storage, but have different policies. Files are organized by media type, and under scoped storage developers do not have anymore the possibility to perform direct file access. Read and write operations are in fact only allowed through MediaStore API calls or Storage Access Framework. Furthermore, files here are permanent, even after the owner app is uninstalled.

For all these reasons, external storage is considered to be a less secure location than internal storage, where stricter enforcing takes place. As usual, developers are invited to make good use of the two alternatives, according to their specific needs.

### 4.1.3 INSTALLATION DIRECTORY

The installation directory is the location where an app's compiled code and native libraries are stored by the system after the APK is installed. Each app has its own folder in the format of

```
data/app/<random_string>
```

where the obfuscated `<random_string>` is generated for security and performance purposes.

An example is

```
/data/app/~~75e0ySpWzTeWghJ9DxLvBw.
```

These folders contain:

- The app's `.apk` file;
- Optimized `.dex` files;
- Native libraries, located in `.so` files.

### 4.1.4 SYSTEM DIRECTORIES

A special location on the storage are system directories. System directories contain system files, and mostly allow only for read operations. In fact, root access is required to modify system files. However, system directories can cover important roles also for the correct functioning of third-party apps.

One example is OpenCamera, which is an open source Android camera app. As any camera app, when pressing the shutter button, a typical sound resembling the physical shutter of old cameras is played by the speaker. That sound is actually a system sound located in `/system/media/audio/ui/camera_click.ogg`. More files are available under the `system` folder, allowing for third party apps to offer a native experience with shared resources.

In Figure 3 are shown the common system directories in Android devices. It is worth noting that the system directories closely resemble those of Linux, further highlighting that the Android kernel is derived from the Linux kernel.

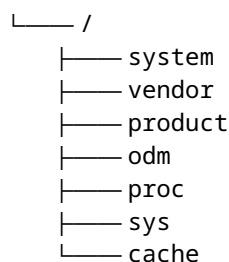


Figure 3 – System directories.

### 4.2 MAKE USE OF THE SCOPED STORAGE

Under scoped storage, we can distinguish three different approaches to access files. Each approach has unique features and limitations. The methods are:

1. **Direct file access**
2. **MediaStore API**
3. **Storage Access Framework**

#### 4.2.1 DIRECT FILE ACCESS

*Direct file access* is the lower level approach of the three. It consists in the usage of the Java `File` library, allowing direct access to a specific file.

In normal conditions, developers have access to internal and external app-specific directories via direct file access.

- External: `storage/emulated/0/Android/data/com.example.packageName/`
- Internal: `data/user/0/com.example.packageName`

```
File file = new File(context.getExternalFilesDir(null), "file.txt");
```

Listing 4 – Example of a `java.io.File` call, pointing to external storage.

The sandboxing rules are strictly enforced: if a file path points to an invalid location, such as another app's internal storage, access to the file is denied. These policies represent the core strength of the Android sandbox model.

#### 4.2.2 MEDIASTORE API

*MediaStore* is the current method of managing media files on the external storage. It is currently the only Google Play compliant way of accessing directories like

```
/storage/emulated/0/DCIM  
or  
/storage/emulated/0/Music.
```

*MediaStore* is internally managed by a database. In fact, the way to insert or obtain media files, as shown in Listing 5, is by using a `ContentResolver`.

```
String[] projection = { MediaStore.Images.Media._ID };
String sortOrder = MediaStore.Images.Media.DATE_ADDED + " DESC";

ContentResolver contentResolver = context.getContentResolver();
Cursor cursor = contentResolver.query(
    MediaStore.Images.Media.EXTERNAL_CONTENT_URI,
    projection,
    null,
    null,
    sortOrder
);

if (cursor != null) {
    // retrieve data from the cursor
    cursor.close();
}
```

Listing 5 — Example of a MediaStore call that obtains all image IDs in descending order.

MediaStore has optional permission requirements. For insertions, any app can always write files on the MediaStore, even without any permission. This is because inserting new files is not considered to be a dangerous action security-wise.

When a MediaStore entry is added to the database, the mandatory primary key is generated and stored on the `BaseColumns._ID` column. The primary key is a progressive integer, starting from 0 onwards and is automatically assigned at creation time. Moreover, it is not possible to modify its value after the entry is saved or before a new addition, as any such operation will be ignored.

Another crucial column in a MediaStore entry is `MediaStore.MediaColumns.OWNER_PACKAGE_NAME`. The value is again assigned automatically and cannot be altered, and is equal to the package name that ultimately completed the insertion.

The ownership information allows the system to determine which app has the authority over a given media item. This is specifically used to determine which files are shown when the MediaStore is queried.

In fact, if the app lacks the permission corresponding to the queried media type, only files owned by the app are returned. If the runtime check confirms the presence of the correct permission, then all files accessible through the MediaStore are returned.

For example, let's consider `MediaStore.Audio`. This content provider is responsible for managing audio files, and is accessed using the URI returned by the code in Listing 6.

```
MediaStore.Audio.Media.getContentUri(MediaStore.VOLUME_EXTERNAL)
```

Listing 6 — Call to get the URI for MediaStore.Audio ContentProvider.

If `READ_MEDIA_AUDIO` permission is denied for the current session, any query to the Media-Store will only show files that are owned by the current package name. Otherwise, any file is queried.

### 4.2.3 STORAGE ACCESS FRAMEWORK

Storage Access Framework, SAF in short, is the most advanced and secure method to access to external storage, specifically tailored for documents. It is presented as a file picker, allowing users to selectively choose the files that the app is permitted to access. This allows for the user to be aware of which files the app can have access to, every time the choice is presented. A basic SAF call is shown in Listing 7.

```
Intent intent = new Intent(Intent.ACTION_OPEN_DOCUMENT);
intent.setType("*/*");
startActivityForResult(intent, REQUEST_CODE);
```

Listing 7 – Example of a SAF call.

After the choice is confirmed, SAF will return to the calling activity a result, containing the URI of the required resource. An example is shown in Listing 8.

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent intent) {
    if (requestCode == REQUEST_CODE && resultCode == Activity.RESULT_OK) {

        Uri uri = intent.getData();
        // access to URI resource
    }
}
```

Listing 8 – Example of a SAF callback.

### 4.3 SANDBOX VULNERABILITIES IN VIRTUALAPP

To perform tests on possible sandbox vulnerabilities, a basic test app was developed. The test app consists of two sections. The first section allows adding and querying elements to and from the MediaStore. The second section is a basic file manager, that allows the user to start exploring the file system from `context.getDataDir()`. Two mirror copies of the test app were then created, one with package name `com.example.filetestred` and one with package name `com.example.filetestblue`. This approach was used to evaluate the sandbox within VirtualApp. By having two different packages, it is possible to check whether the sandbox rules are enforced, and up to which point they break.

Shown in Figure 4, each app has a colored UI that matches the package name, to allow for easier recognition. Moreover, each app has a `.txt` file located on the root of the internal storage. This file will be used to test the ability to read other app's owned resources.



Figure 4 — `com.example.filetestred` and `com.example.filetestblue`, on the file manager. Each of the two apps points to their respective internal storage, on the redirected path.

### 4.3.1 DIRECT FILE ACCESS - VIRTUALAPP

As previously discussed, VirtualApp manages the file system redirection. This ensures that virtual applications have access to a clean and stock-like file system. The redirection is targeted to a specific folder inside the internal storage of VirtualApp.

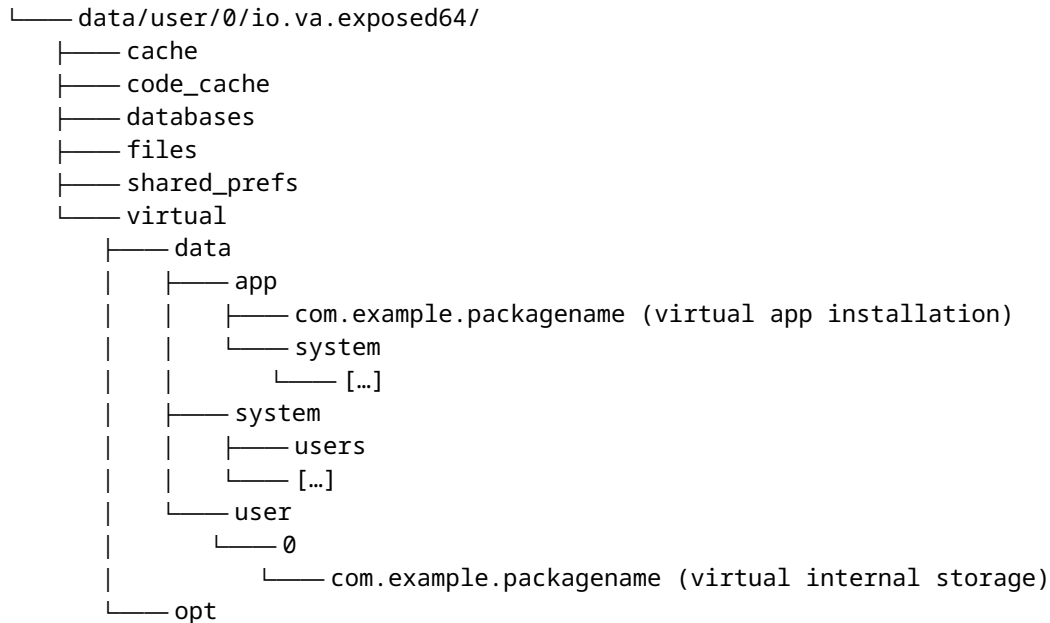


Figure 5 — Overview of the container’s private storage. The virtual directory holds the virtual file system.

The folder `data/user/0/io.va.exposed64` shown in Figure 5 is the private, sandboxed directory of VirtualApp. More specifically, it is the private sandboxed folder of the app whose package name is `io.va.exposed64`. This folder is only accessible within the VirtualApp application itself, and no other app can read nor write data to it. This property is granted by the sandbox model, and such strict policy is what keeps Android strongly tailored to security.

VirtualApp defines a `virtual` folder, where the file system redirection happens. That directory contains a simplified representation of the Android file system.

Two fundamental folders are the following:

- `virtual/data/app/`, where are redirected all the installation directories of virtual apps;
- `virtual/data/user/0/`, where are redirected all the sandboxed directories of each virtual app. In other words, that is where each virtual app’s internal storage is located.

The redirection takes place whenever the plugin app calls `context.getFilesDir()`. Plugin apps are technically run under the container app’s space, so the call of this method is ultimately executed by the latter. That method call should be expected to return in fact `data/user/0/io.va.exposed64`, which will surely cause severe functioning problems. Through the redirection, the method call is intercepted, and the return value is altered to match `data/user/0/io.va.exposed64/virtual/data/user/0/com.example.packageName`. This

allows for the plugin app to have a clean and stock-like internal storage. Every plugin app has its own internal storage assigned, and everything works as expected.

However, there is a severe issue. Plugin apps are not physically installed: they are rather directly executed by loading the classes from `.dex` files at runtime. This means that from Android's perspective, the app which is being executed is still and will always be the container app. This means that all plugin apps inherit the sandboxing permissions from the container. As a consequence, all plugin apps have read and write access to the entire directory `data/user/0/io.va.exposed64` and all the children directories.

This is a severe vulnerability, as the sandbox property is broken. Plugin apps can have mutual access to private files. The attack is shown in Figure 6.

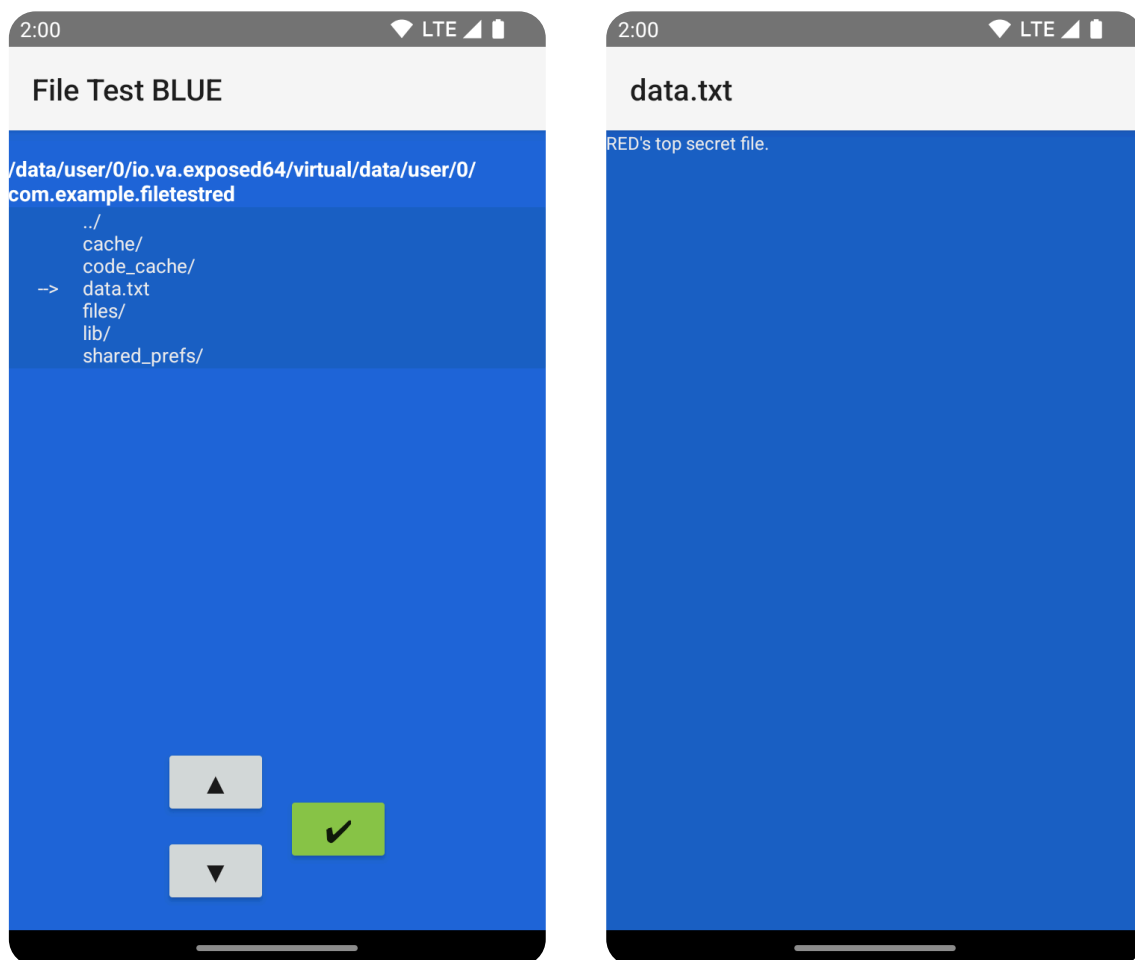


Figure 6 — Blue is capable of backtracking on the directories tree and open Red's private file.

VirtualApp stores configuration files in directory `/data/user/0/io.va.exposed64/virtual/data/app/system`. Being part of the internal storage, the configuration files stored in that directory are also accessible to plugin apps.

### 4.3.2 MEDIASTORE - VIRTUALAPP

The main vulnerability concerning MediaStore lies in its permission management. For the proper functioning of the framework, the container app declares all permissions on the manifest file, and the user must grant them all. This means that all the variants of the `READ_MEDIA_...` permissions are granted. All plugin apps then inherit the permissions from the container, and are able to always access to all files, even the ones that are not owned by the current plugin app.

The permission vulnerability on VirtualApp was studied and solved by Alberto Lazari, as he carefully explained that permissions support must be implemented manually in order to enable the correct enforcing. It was a rather simple matter to add support for MediaStore permissions, but still MediaStore was not working as it should. This is because a deeper problem was hiding.



Figure 7 — Red creates a media audio file named RED.mp3. Blue is capable of querying also files created by Red, even if the permission `Manifest.permission.READ_MEDIA_AUDIO` is denied.

As discussed before, plugin apps are executed by the system within the container. This is also reflected on MediaStore insertions. For any new insertion on the MediaStore, the system assigns to `MediaStore.MediaColumns.OWNER_PACKAGE_NAME` column the package name of the container. This means that all plugin apps insertions share the same owner, which is the container app. The issue is that MediaStore relies on the `OWNER_PACKAGE_NAME` column to determine the file ownership, returning all files even if the permission is denied. The attack is shown in Figure 7.

### 4.3.3 STORAGE ACCESS FRAMEWORK - VIRTUALAPP

VirtualApp also supports SAF calls. However, since SAF is managed on the UI by a file picker, there is precise intent for the user to open a specific file. Because of this, SAF calls do not pose any vulnerability, as the access is granted for one time and only for the picked resource.

# 5

## SECURING DIRECT FILE ACCESS

### 5.1 GENERAL OVERVIEW

Direct file access is performed in Listing 9.

```
file = new File(context.getExternalFilesDir(null), "file.txt");
```

Listing 9 – Direct file access call.

What is happening behind the scene is a series of method calls in the Android framework, each one to lower level layers than the previous.

Here is the succession of calls, taken directly from the Android source code:

```
public FileInputStream(File file) {  
    // ...  
    fd = IoBridge.open(name, O_RDONLY);  
    // ...  
}
```

Listing 10 – Constructor in FileInputStream.java [23]

## 5. SECURING DIRECT FILE ACCESS

---

```
public static FileDescriptor open(String path, int flags) {
    FileDescriptor fd = null;
    try {
        fd = Libcore.os.open(path, flags, 0666);
        // ...
    }
}
```

Listing 11 — Method `open()` in `IoBridge.java` [24]

```
public FileDescriptor open(String path, int flags, int mode) {
    return os.open(path, flags, mode);
}
```

Listing 12 — Method `open()` in `ForwardingOs.java` [25]

```
public native FileDescriptor Linux.open(String path, int flags, int mode);
```

Listing 13 — Method `open()` in `Linux.java` [26]. Note that `Linux` extends `Os`.

`os.open()` is the last Java method to be called in the stack. Moreover, it happens to be the last Java method called in any file-related access. For example, any `SharedPreferences` access has the following call stack:

```
└── SharedPreferencesImpl.EditorImpl.commit()
    └── SharedPreferencesImpl.enqueueDiskWrite()
        └── SharedPreferencesImpl.writeToFile()
            └── FileOutputStream()
                └── IoBridge.open()
                    └── Os.open()
                        └── native Linux.open()
```

Figure 8 — Stack of method calls, from `commit()` in `SharedPreferencesImpl.java`.

For this reason, a `Proxy` is set for the method `os.open()`. Anytime a plugin app requests a file access, the `InvocationHandler` interface linked to the `Proxy` is executed, leading to execution of the modified implementation of the method.

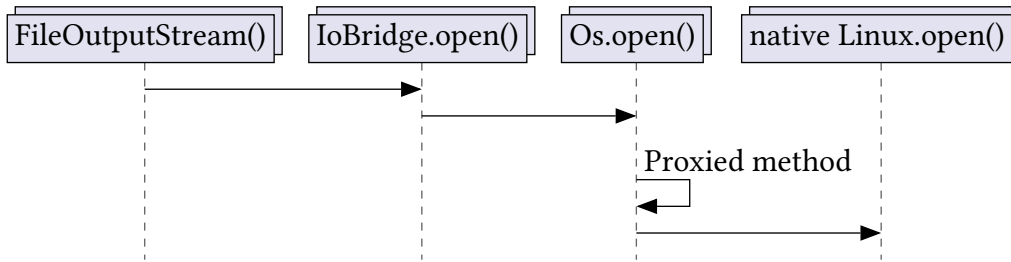


Figure 9 – Sequence of method calls for a file access. The proxy is placed in `libcore.io.Os.open()`.

As shown in Figure 9, the proposed implementation places a proxy on the method `Os.open()`. Inside the `Proxy` implementation, it is sufficient to check at runtime the requested path. The method `Os.open()` already has useful parameters:

- `String path`: path of the file to be opened;
- `int flags`: bitmask of the access, file creation and file status flags;
- `int mode`: the file mode bits to be applied when a new file is created.

The implementation checks the parameter `path`, and compares it to a series of policies. The policies are then used to determine whether the file access should be granted or not.

Let’s assume that a plugin app with package name `com.example.myapplication` is performing file access. Its dedicated virtual internal storage is located at `data/user/0/io.va.exposed64/virtual/data/user/0/com.example.myapplication`. This means that the policy should grant access to that private virtual directory only to virtual package `com.example.myapplication`.

Another package `com.example.malicious` must not have access to the same directory. Because of this, the `Proxy` implementation determines at runtime the package name and the virtual user of the requesting plugin app. It then uses the policies to determine whether the request is valid or not.

## 5. SECURING DIRECT FILE ACCESS

---

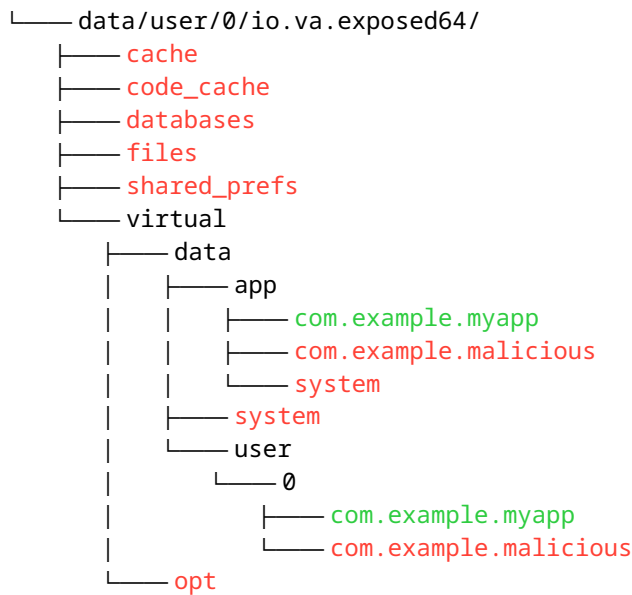


Figure 10 – The policies applied for the directories included in `data/user/0/io.va.exposed64/`, relative to package name `com.example.myapp` and user `0`. Green indicates that the access is permitted, red blocked.

The custom implementation only enforces the policies inside the container’s internal and external storage. This is because the sandbox violations only happen within the virtual storage of plugin apps.

For example, if an app tries to access to `/system/media/audio/ui/camera_click.ogg`, which is located in a system directory, it should be able to access regardless of the calling package name. This is because system directories are outside the sandbox scope, and are protected by the Linux file access policies. In fact, system directories never allow for write operations, and some like `/system/media/` allow for read operations. In a sense, giving free access to these directories does not mean that plugin apps have full potential over those locations, rather we are delegating file access control policies to Linux.

## 5.2 KEY COMPONENTS

### 5.2.1 PATH TOKENIZER

`PathTokenizer` is the component responsible for splitting an absolute path into tokens. Tokens are stored in String arrays, and each one represent one level of the directories tree for a path. For example, the path `/data/user/0/com.example.packagename` is represented as:

data	user	0	com.example.packagename
------	------	---	-------------------------

`PathTokenizer` is defined as a static class inside `PathChecker`, and all methods are shown in Listing 14.

```
private static final class PathTokenizer {  
  
    public static String[] getTokens(String path) {  
        return path.substring(1).split("/");  
    }  
  
    public static String getPath(final String[] tokens) {  
        StringBuilder result = new StringBuilder();  
        for(String token : tokens) {  
            result.append("/").append(token);  
        }  
        return result.toString();  
    }  
}
```

Listing 14 — `PathTokenizer` class.

Splitting the string path into tokens allows for easier usage in comparison algorithms.

***GetTokens.*** This method takes as input a String representing an absolute path and returns an array of Strings, each one representing a token.

***GetPath.*** From an array of Strings representing tokens, this method builds a String representing the absolute path.

### 5.2.2 PATHCHECKER

PathChecker is a class that is responsible for checking whether the current path is valid for the calling plugin app. The policies are defined here.

PathChecker has the following model for the policies: **allow all**, with **blacklist** and **exceptions**.

- The **blacklist** allows for blocking access to the entire `/data/user/0/io.va.exposed64` directory;
- The **exceptions** allow for granting access only to the selected directories within `/data/user/0/io.va.exposed64`;
- **Allow all** is to grant access to any other directory outside the blacklist, that will result in a fallback to the Linux protection mechanisms.

A series of placeholders are defined. These placeholders are dynamically substituted at runtime with the correct values for the plugin app being executed.

- `^USER^`: system user that owns the container;
- `^HOST_PACKAGE^`: package name of the container;
- `^VIRTUAL_USER^`: virtual user associated to the plugin app in execution;
- `^VIRTUAL_PACKAGE^`: virtual package name associated to the plugin app in execution;
- `^INSTALL_DIR^`: installation directory of the container.

The default policies are shown in Listing 15.

```
private static final String[] exceptions = {
    "/data/user/^USER^/^HOST_PACKAGE^/virtual/data/user/^VIRTUAL_USER^/
^VIRTUAL_PACKAGE^/*",
    "/data/user/^USER^/^HOST_PACKAGE^/virtual/data/app/^VIRTUAL_PACKAGE^/*",
    "/data/user/^USER^/^HOST_PACKAGE^/virtual/data/user/^VIRTUAL_USER^/
wifiMacAddress",
    "/storage/emulated/^USER^/Android/data/^HOST_PACKAGE^/virtual/^VIRTUAL_USER^/
^VIRTUAL_PACKAGE^/*"
};

private static final String[] blacklist = {
    "/data/user/^USER^/^HOST_PACKAGE^/*",
    "^INSTALL_DIR^/*",
    "/storage/emulated/^USER^/Android/data/^HOST_PACKAGE^/*"
};
```

Listing 15 — Base policies with placeholders.

The symbol `*` at the end of a path means that that policy has to be applied to the specified directory and any of the children directories.

In order to speed up the path verification, a cache is used. This is especially useful when accessing recurrent files. A specific evaluation is performed in subsection 7.3.1.

**Blacklist.** The paths represented here will be considered blacklisted, and access will be strictly forbidden. The paths here are the internal and external storage of the container, and its installation folder.

**Exceptions.** There are four exceptions on the policy:

1. Virtual internal storage of the plugin app under execution;
2. Virtual external storage of the plugin app under execution;
3. Virtual installation folder of the plugin app under execution;
4. A file named `wifiMacAddress` that is frequently accessed by the framework and does not pose any security concern.

**Microsoft AppCenter.** The `com.microsoft.appcenter.AppCenter` library is part of Microsoft App Center SDK, which developers use to integrate Microsoft App Center services into their Android apps. In the case of VirtualXposed, this library is used only to collect analytics about crashes. This library uses shared preferences to store some values in `/data/user/0/io.va.exposed64/shared_prefs/AppCenter.xml` file. This file does not certainly contain sensitive information, and technically could be added to the exceptions, exactly like `wifiMacAddress`. However, being just a third party analytics tool, it was decided to completely disable its access. By doing so there is not any downside in terms of functionality.

**wifiMacAddress.** From a technical point of view, the access to this file could also be disabled. Thorough testing was conducted, and disabling this file does not impact the functionality of the virtualization. However, this file is directly part of the framework, unlike `AppCenter.xml` that is from a third party optional library. For this reason, it was decided to keep it available, as the data stored is not sensible.

PathChecker is instantiated in Listing 16.

```
PathChecker pathChecker = PathChecker.get();
```

Listing 16 — Instantiation of PathChecker.

The constructor calls the method `replace()`, shown in Listing 17, which retrieves all the values for each placeholder, and substitutes them to the policies.

```
private String replace(String original) {
    String installDir = // Retrieve the runtime value...
    String systemUser = // ...
    String hostPackage = // ...
    String virtualUser = // ...
    String virtualPackage = // ...

    original = original.replace("^INSTALL_DIR^", installDir);
    original = original.replace("^USER^", systemUser);
    original = original.replace("^HOST_PACKAGE^", hostPackage);
    original = original.replace("^VIRTUAL_USER^", virtualUser);
    original = original.replace("^VIRTUAL_PACKAGE^", virtualPackage);

    return original;
}
```

Listing 17 — The `replace()` method substitutes the placeholders with runtime values.

The updated policies with true absolute paths are then tokenized and stored on the class.

***isPathValid.*** This is the method that is called to check the validity of a designated path. The logic, shown in Listing 18, can be broken into four sections. The method:

1. Checks whether the path has already been evaluated and stored on the cache. If that is true, immediately return the pre-calculated policy.
2. If not already checked, checks whether the path is among the exceptions. If that is true, it means that the path is either the virtual installation directory, virtual internal storage or virtual external storage. In any case, the access should be granted.
3. If not on the exceptions, checks whether the path is among the blacklist. If that is true, it means that the path is pointing to the internal storage of the container. The access must be denied.
4. Else, grant unconditionally. The path is pointing to any location that is not covered by the sandbox. The Linux enforcing will take place.

```
public boolean isPathValid(String path) {
    if(!isInit) init();

    // 1) Check cache
    if(cache.containsKey(path)) return Boolean.TRUE.equals(cache.get(path));

    final String[] tokens = PathTokenizer.getTokens(path);

    // 2) Check if accessed path is among the exceptions
    for(String[] list : exceptionsPaths) {
        int i = 0;
        while(i < list.length && i < tokens.length) {
            if(list[i].equals("*") && tokens.length >= list.length) return true;
            if(!tokens[i].equals(list[i])) break;
            if(i == list.length - 1 && i == tokens.length - 1 &&
                tokens[i].equals(list[i]))
                return true;
            i++;
        }
    }

    // 3) If not, check if it's blacklisted
    for(String[] list : blacklistPaths) {
        int i = 0;
        while(i < list.length && i < tokens.length) {
            if(list[i].equals("*") && tokens.length >= list.length) return false;
            if(!tokens[i].equals(list[i])) break;
            if(i == list.length - 1 && i == tokens.length - 1 &&
                tokens[i].equals(list[i]))
                return false;
            i++;
        }
    }
    return true; // 4) Finally, grant unconditionally
}
```

Listing 18 — `isPathValid()` method checks the path validity.

The `init()` method populates all policies, by substituting each placeholder with the corresponding runtime value. It is only computed once, since for each file access the values are persisted.

### 5.2.3 LIBCORESTUB

`LibCoreStub` is the component of the framework that is responsible for proxying methods of the package `libcore.io`. A new implementation of the proxied method `open()` is required. Given the extensive usage of proxies in `VirtualApp`, the developers implemented a very convenient helper class, `MethodProxy`, and a model that allows for creating and

installing proxies with ease. The `MethodProxy` class also allows for calling utility methods directly from the class, making it the perfect choice for the task.

The proxy for the method `open()` is defined in Listing 19.

```
public static class Open extends MethodProxy {
    @Override
    public String getMethodName() {
        return "open";
    }

    @Override
    public Object call(Object who, Method method, Object... args) {
        String path = (String) args[0];
        if(VirtualRuntime.getInitialPackageName() != null) {
            PathChecker checker = new PathChecker();
            if(checker.isPathValid(path)) {
                // Valid path: return base open() implementation
                return method.invoke(who, args);
            }
            // Invalid path: block file access
            return null;
        }

        return method.invoke(who, args); // Fallback to base implementation
    }
}
```

Listing 19 — `open()` `MethodProxy`.

The original signature of the `Os.open()` method, taken from the Android source code, is shown in Listing 20.

```
public FileDescriptor open(@Nullable String path, int flags, int mode)
```

Listing 20 — Original signature of the method `Os.open()`.

The `Proxy` intercepts the first original parameter `@Nullable String path`, and instantiates a new `PathChecker` object. The constructor will retrieve the current values for the placeholders of the plugin app in execution. Then, `PathChecker.isPathValid(path)` is called, and it will return whether the path is valid or not, according to the current plugin app. If the path is not valid, `null` will be returned. A safe fallback is also implemented just for precaution. The safe fallback is intended to be executed only when the framework could not find the correct runtime values for the placeholders. In the implementation, it was decided to fall back to the base implementation, which in turn translates to granting the access. It is possible to turn it to a more conservative approach, where the fallback implementation returns `null`, effectively denying the access.

# 6

## SECURING MEDIASTORE

### 6.1 GENERAL OVERVIEW

For new MediaStore file insertions, some columns are user-settable.

```
String fileName = "RED.mp3";

ContentValues values = new ContentValues();
values.put(MediaStore.Audio.Media.DISPLAY_NAME, fileName);
values.put(MediaStore.Audio.Media.MIME_TYPE, "audio/mp3");
values.put(MediaStore.Audio.Media.RELATIVE_PATH, Environment.DIRECTORY_MUSIC);

// Insert the value on the MediaStore Content Provider
// ...
```

Listing 21 – Basic MediaStore insertion. Some of the columns are set by calling put().

This is performed with a ContentValues object. Among the settable columns, we have:

- DISPLAY\_NAME: the filename to be created.
- MIME\_TYPE: the media type. The value must be consistent to the actual type of file.
- RELATIVE\_PATH: The path where the file is saved. This also needs to be consistent, e.g. store images under DIRECTORY\_IMAGES, audio under DIRECTORY\_MUSIC etc.
- IS\_PENDING: sets the pending state to true.

## 6. SECURING MEDIASTORE

---

Other columns are automatically set by the OS. Among those, we have:

- `_ID`: the unique ID of the file. This is the Primary Key of the database.
- `SIZE`: the file size.
- `OWNER_PACKAGE_NAME`: the package name that completed the file insertion.

For the code in Listing 21, run by natively installed APKs, the `MediaStore.Audio` database after the insertion will look like Table 3.

<code>_ID</code>	<code>DISPLAY_NAME</code>	<code>RELATIVE_PATH</code>	<code>OWNER_PACKAGE_NAME</code>	...
1	RED.mp3	storage/emulated/0/Music	com.example.filetestred	...

Table 3 — Representation of a MediaStore entry.

The value for column `OWNER_PACKAGE_NAME` has been automatically set by the OS, matching the package name that completed the file insertion.

Let's assume that `com.example.filetestblue` inserts a `BLUE.mp3` file, and `com.example.filetestred` inserts a new `RED.mp3` file. The database will look like Table 4.

<code>_ID</code>	<code>DISPLAY_NAME</code>	<code>RELATIVE_PATH</code>	<code>OWNER_PACKAGE_NAME</code>	...
1	RED.mp3	storage/emulated/0/Music	com.example.filetestred	...
2	BLUE.mp3	storage/emulated/0/Music	com.example.filetestblue	...
3	RED(1).mp3	storage/emulated/0/Music	com.example.filetestred	...

Table 4 — Representation of three MediaStore entries.

Let's assume that `com.example.filetestred` does not have access to the `READ_MEDIA_AUDIO` permission, and tries to select all files from the database, by calling the code on Listing 22.

```
Cursor cursor = contentResolver.query(
    MediaStore.Audio.Media.EXTERNAL_CONTENT_URI,
    null, // projection (null = all)
    null, // selection (null = all)
    null, // selectionArgs
    MediaStore.Audio.Media.DATE_ADDED + " DESC" // sort by most recent
);
```

Listing 22 — Example query.

MediaStore will restrict only to files that have `com.example.filetestred` as package name.

The returned table will look like Table 5.

## 6. SECURING MEDIASTORE

---

_ID	DISPLAY_NAME	RELATIVE_PATH	OWNER_PACKAGE_NAME	...
1	RED.mp3	storage/emulated/0/Music	com.example.filetestred	...
3	RED(1).mp3	storage/emulated/0/Music	com.example.filetestred	...

Table 5 — Returned table. Files owned by `com.example.filetestblue` are not visible.

If instead we give grant access to the respective permission `READ_MEDIA_AUDIO`, the entirety of the rows will be returned.

When running these two test apps on the virtualization framework, we first see the issue: the insert transaction is completed by the container app. Thus, the database will look like this Table 6.

_ID	DISPLAY_NAME	RELATIVE_PATH	OWNER_PACKAGE_NAME	...
1	RED.mp3	storage/emulated/0/Music	io.va.exposed64	...
2	BLUE.mp3	storage/emulated/0/Music	io.va.exposed64	...
3	RED(1).mp3	storage/emulated/0/Music	io.va.exposed64	...

Table 6 — Representation of three MediaStore entries, when inserting from plugin apps.

All files are owned by the container. Thanks to the work by Alberto Lazari, with virtual permissioning it is possible to enforce checks for any permission, given a custom implementation.

Obviously the `OWNER_PACKAGE_NAME` column is not writable. A custom virtual ownership model is required. Such model should work on top of the existing ownership, and be available to plugin apps.

When a new file is added to the MediaStore, the virtual ownership model adds a record to a table. The table, informally named *virtual ownership table* and depicted in Table 7, links the `_ID` column value to the virtual UID that executed the call.

_ID	VIRTUAL_UID
1	10002
2	10003
3	10002

Table 7 — Virtual ownership table.

In this example, virtual UID 10002 is assigned to plugin app `com.example.filetestred` and virtual UID 10003 to `com.example.filetestblue`. The VUIDs are generated by `VirtualApp` and remain consistent throughout all the lifecycle of the plugin app. `VirtualApp`, in fact,

has a method that is used to uniquely compute the value for the VUID given virtual user and package name.

The implementation for the virtual UID generation is reported on Listing 23 from the class `VUserHandle`:

```
public static int getUid(int userId, int appId) {
    if (MU_ENABLED) {
        return userId * PER_USER_RANGE + (appId % PER_USER_RANGE);
    } else {
        return appId;
    }
}
```

Listing 23 — Method used by VirtualApp to obtain VUIDs.

The virtual ownership table is all it is needed to allow for linking the MediaStore files to their respective owner.

In order to restrict the select operation to only owned files, an interesting approach is used. At runtime, the permission for the specific media type is checked: if the permission is granted, we do not need any further action as it is correct to select all files from all package names, comprising all plugin apps. If the respective permission is not granted instead, the query is intercepted via a Java Proxy, and altered in such a way to only select for the correct virtual UID.

Let's make an example using SQL language. We query the MediaStore for all audio files from a plugin app with virtual UID 10002. The respective SQL statement, that is generated under the hood, is the following:

```
SELECT * FROM media_audio;
```

This statement returns all entries on the database. Assume that we do not want to access to files owned by other virtual apps.

The framework needs to access the virtual ownership table, looking for matches for virtual UID 10002. Then, it should inject a `WHERE` clause, with as arguments of selection the IDs that were found. The statement then becomes:

```
SELECT * FROM media_audio WHERE _ID IN (1, 3);
```

This dynamic approach allows for selecting only owned files.

This approach needs to be used also when trying to remove a file. We can only remove files that are not owned if we have grant access to the permission. The same approach as the select takes place.

```
DELETE FROM media_audio;
```

becomes

```
DELETE FROM media_audio WHERE _ID IN (1, 3);
```

## 6.2 KEY COMPONENTS

### 6.2.1 MEDIASTORE FILE PARSER

The virtual ownership table is stored onto a `.json` file. Such format was chosen primarily for the convenience of libraries that manage `.json` files, compared to other formats like `.xml` that require more complicated and less readable code. Moreover, for this usage, `.json` files are more than enough since we do not need a complicated hierarchy between elements, just (ID, UID) pairs.

The `MediaStoreFileParser` class allows for parsing a `media.json` file, to reconstruct the cache.

```
public void write(final Map<Integer, Integer> cache, final OutputStream fileStream)
{
    final JSONArray jsonArray = new JSONArray();
    for(int key : cache.keySet()) {
        final JSONObject jsonObject = new JSONObject();
        jsonObject.put(ID, key);
        jsonObject.put(OWNER, cache.get(key));

        jsonArray.put(jsonObject);
    }
    final ByteArrayOutputStream byteStream = new ByteArrayOutputStream();
    byteStream.write(jsonArray.toString().getBytes());
    byteStream.writeTo(fileStream);
}
```

Listing 24 — Code for a write operation under `MediaStoreFileParser`. The content of the cache is read, and a `JSONArray` object is created. Then, the file is overwritten with the updated content.

Listing 25 shows a sample of how the data is stored on the `.json` file.

```
[
  {"id":1000003586,"owner":10002},
  {"id":1000003584,"owner":10002},
  {"id":1000003585,"owner":10002}
]
```

Listing 25 — Sample of how data is stored on `.json` files.

For each stored entry, there is information about the unique ID of the resource and the virtual UID.

### 6.2.2 MEDIASTORE CACHE

The `MediaStoreCache` class is responsible for representing an instance of the virtual ownership table. It is built on top of the class `LockedOperation`. Such class was coded by Alberto Lazari, and offers utility methods to safely read and write to a file, especially from the perspective of concurrent access.

```
private final MediaStoreFileParser parser;  
private Map<Integer, Integer> cache;
```

Listing 26 — Parser and cache on `MediaStoreCache` class.

As shown in Listing 26, the class contains an instance of a `MediaStoreFileParser`, that will be used to parse the `.json` file. Moreover, a `Map<Integer, Integer>` is instantiated. This is the actual cache, that holds the pair (`ID`, `UID`) for each `MediaStore` entry.

The cache will be refreshed from storage only when needed. This makes each read operation much faster than reading every time from storage, and is crucial to keep a stable performance on the run.

### 6.2.3 VIRTUAL MEDIA PROVIDER

The `VMediaProvider` class is responsible for the implementation of the whole virtual ownership model.

It is modeled as a singleton, so its instance can be recovered anywhere within the process. It also defines three caches, one for each media type, as shown on Listing 27.

```
private static MediaStoreCache audioCache;  
  
private static MediaStoreCache videoCache;  
  
private static MediaStoreCache imageCache;
```

Listing 27 — The declaration of three `MediaStore` caches.

The `.json` files are located on `data/user/0/io.va.exposed64/virtual/data/app/system/mediastore/`

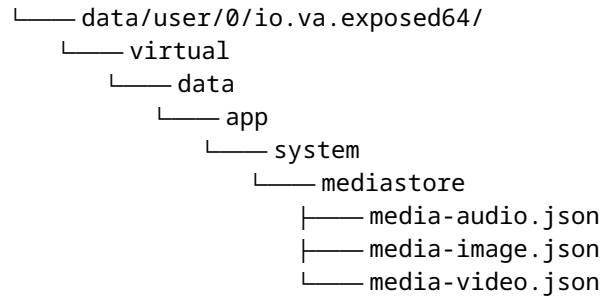


Figure 11 – Overview of the MediaStore-related .json files.

The methods that `VMediaProvider` offers are the following:

- `void add(int id, int owner, Uri uri)`: adds a new entry on the selected virtual ownership table;
- `Integer[] getIds(final int owner, final Uri uri)`: gets the IDs for a virtual UID;
- `int getOwner(final int id, final Uri uri)`: gets the virtual UID given a file ID;
- `void remove(final int id, final Uri uri)`: removes the entry from the cache, by the ID.
- `void removeOwner(final int owner, final Uri uri)`: removes all entries that are owned by a specific virtual UID;
- `void unown(final int owner)`: substitutes the owner value to -1, which is interpreted as *owned by the framework*.

Upon creation of a new instance, the constructor executes a routine that checks whether files have been altered meanwhile the framework was offline. To explain this case, let's assume that we have a couple of files stored on the MediaStore, owned by plugin apps. This means that the virtual ownership table is populated with the respective pairs.

It is possible for apps to remove files that are owned by other apps, given the appropriate permission. The stock file manager is, in fact, an app with the appropriate permission and thus can explore and delete files. If this happens, we encounter a problem: the reference is still present on the virtual ownership table. The reference must be removed to grant the efficiency and good state of the model. Technically speaking, security is also a concern. This is because IDs are progressive integers and in the real case it is near to impossible to reach the maximum number and loop back from 0 to the phantom entry. However, a malicious app could perform looped insertion to exhaust all IDs and loop back from 0, though this offers very little room for a real use case.

The `detectChanges()` routine calls and run its code from a `Service`. This happens because this code is intended to be run from the framework's main process, and not from any plugin apps' dedicated one. The specific reason is explained in the following subsection.

## 6.2.4 MEDIA PROVIDER SERVICE

This service is intended to perform routine operations on the cache.

```
public int onStartCommand(Intent intent, int flags, int startId) {
    int mode = intent.getIntExtra(MODE, -1);
    switch(mode) {
        case MODE_DETECT-> detectChanges(intent);
        default -> throw new RuntimeException();
    }

    stopSelf();
    return super.onStartCommand(intent, flags, startId);
}
```

Listing 28 — Method `onStartCommand()` in `MediaProviderService`.

As shown in Listing 28, the service checks the flag that is embedded on the `Intent` upon the Service creation, and starts the appropriate routine. Up to now, it only supports the `MODE_DETECT` operation, which is used during framework bootup to refresh the virtual ownership table.

```
private void detectChanges(final Intent intent) {
    Uri uri = intent.getParcelableExtra(URI);
    VMediaProvider provider = VMediaProvider.get();

    // Get all ids (owned by all plugin apps with -1)
    Integer[] ids = provider.getIds(-1, uri);
    if(ids == null) return;

    String[] projection = new String[]{BaseColumns._ID};
    String selection = MediaStore.MediaColumns.OWNER_PACKAGE_NAME + " == ?";
    String[] selectionArgs = new String[]{VirtualCore.get().getHostPkg()};
    Cursor cursor = VirtualCore.get().getContext().getContentResolver().query(
        uri,
        projection,
        selection,
        selectionArgs,
        MediaStore.Audio.Media.DATE_ADDED + " DESC"
    );
    Vector<Integer> vector = new Vector<>();
    if (cursor != null) {
        while (cursor.moveToNext()) {
            int id = cursor.getInt(cursor.getColumnIndexOrThrow(BaseColumns._ID));
            vector.add(id);
        }
        cursor.close();

        for(int id : ids)
            if(!vector.contains(id)) provider.remove(id, uri);
    }
}
```

Listing 29 — Method `detectChanges()` in `MediaProviderService`.

The `detectChanges()` routine shown in Listing 29 follows a straightforward approach.

First, we get all IDs of all files that have been stored by plugin apps on the MediaStore, as shown in Listing 30. This information is located on the virtual ownership table.

```
Integer[] ids = provider.getIds(-1, uri);
```

Listing 30 – Call to get all files that have been stored by plugin apps.

The first parameter represents the virtual UID of the requested app. `-1` as parameter, is interpreted as *owned by the framework*, in other words it allows for selecting entries from any plugin app.

Second, I perform a normal query on the MediaStore, again selecting for all files that are owned by the container app.

In the normal case, these two lists should match. If the IDs list computed on the first step has more entries than the one computed on the second step, it means that those extra files have been removed while the framework was offline. We just need to remove those entries from the table.

The purpose of delegating this code to a Service has to do with how proxies and processes are handled on the virtual environment. When performing the query to the MediaStore database, we want to trigger the original implementation of `query()`. However, the same method is proxied, with a custom implementation showed in subsection 6.2.5. Proxies are valid within the plugin-specific process that is spawned by `VirtualApp`. In order to avoid proxies, it is needed to run the code on another process, where they are not active. If the Service is defined on the manifest of the container, it will run on its main process, rather than running on the plugin's process. By doing so, we avoid triggering the proxied `query()` when calling the method `detectChanges()`. This problem was also relevant on other occasions, explained in subsection 6.2.5 and section 6.3, where other strategies are needed to avoid the triggering of proxies.

### 6.2.5 MEDIA PROVIDER HOOK

Class `MediaProviderHook` is the direct extension of `ProviderHook` and is responsible for redirecting all methods that interact with the MediaStore through a `ContentProvider`. `ProviderHook` is essentially an `InvocationHandler` object, that implements the invocation for a Proxy. Its `invoke()` method is a very long but simple implementation, that parses the parameter `name` to redirect to the desired reimplementaion.

```
public Object invoke(Object proxy, Method method, Object... args) {
    String name = // Get method name
    if ("insert".equals(name)) {
        // Parameter modification
        return insert(methodBox, url, values);
    } else if ("delete".equals(name)) {
        // Parameter modification
        return delete(methodBox, url, selection, selectionArgs);
    } else if ("bulkInsert".equals(name)) {
        // Parameter modification
        return bulkInsert(methodBox, url, initialValues);
    } else if ("update".equals(name)) {
        // Parameter modification
        return update(methodBox, url, values, selection, selectionArgs);
    }
    // ...
}
```

Listing 31 – A simplified representation of `ProviderHook.invoke()`.

The class `MediaProviderHook` extends `ProviderHook` allows for custom implementations of such methods, so that the correct one is always executed based on the type of the `ContentProvider`.

**Insert.** This method is called when a new element is inserted on the `MediaStore`. The logic here is simple: we obtain the value of the `_ID` column and then populate the virtual ownership table with the pair (`ID`, `UID`).

However, things are not straightforward for a specific reason: the column `_ID` is generated by lower level methods in the call stack. This means that when `ProviderHook` is called, the `ID` is not generated yet. A new approach is needed.

On Listing 32 is shown a simplified implementation of the solution.

```
public synchronized Uri insert(MethodBox methodBox, Uri url, ContentValues
initialValues) throws InvocationTargetException {
    // Finish the insert operation
    Uri retVal = super.insert(methodBox, url, initialValues);

    // If the file is not set to pending
    // (pending = 1 hides the file until it's set to 0)
    Object isPending = initialValues.get(MediaStore.MediaColumns.IS_PENDING);
    if(isPending == null || isPending.equals(0))
        VMediaProvider.finalizeInsert(VBinder.getCallingUid(), url);

    return retVal;
}
```

Listing 32 – Method `insert()` in `MediaProviderHook`.

Instead of directly returning the method, we complete the insertion on the `MediaStore` and then save the output to an object named `retVal`. This has the purpose of generating the `_ID` value on its respective column.

After that, a check on the pending state is done. A file can be set to be on pending state during the insertion on the MediaStore. This is a way to safely and atomically insert new media files without immediately making them visible to other apps. The file exists on the MediaStore, but it is not yet marked as visible and thus will not show up to other apps. This is especially useful for video files, where insertions can take more time due to larger file size.

If the file is not set for pending state, or the file was set to pending and then reverted to normal, `VMediaProvider.finalizeInsert()` is called and `retVal` is returned.

Listing 33 shows `finalizeInsert()`, which is a method that checks for the latest file that has been inserted on the MediaStore, and creates a new entry on the virtual ownership table.

```
public synchronized static void finalizeInsert(final int owner, final Uri uri) {
    String[] projection = new String[]{BaseColumns._ID,
    MediaStore.MediaColumns.DATE_ADDED};

    Bundle queryArgs = new Bundle();
    queryArgs.putString(
        "android:query-arg-sql-selection",
        MediaStore.MediaColumns.OWNER_PACKAGE_NAME + " == ?"
    );
    queryArgs.putStringArray(
        "android:query-arg-sql-selection-args",
        new String[] {VirtualCore.get().getHostPkg()}
    );

    ProviderHook.deactivate();
    Cursor cursor = VirtualCore.get().getContext().getContentResolver().query(
        uri,
        projection,
        queryArgs,
        null
    );
    ProviderHook.activate();

    MediaStoreEntry[] entries = MediaStoreCursorParser.parseCursorAndSort(cursor);
    if(entries.length > 0) {
        int id = Integer.parseInt(entries[0].get(BaseColumns._ID));
        VMediaProvider.get().add(id, owner, uri);
    }
}
```

Listing 33 — Method `finalizeInsert()` in `MediaProviderHook`.

It should be noted that the cursor is not directly parsed by a loop. It is rather passed as an argument to a static method defined in class `MediaStoreCursorParser`. This class allows solving a serious issue regarding the timeliness of insert operations on the database. The column `MediaStore.MediaColumns.DATE_ADDED` is responsible for storing the Unix timestamp in seconds. When inserting two or more files within the same second, they all share the

same value for column `DATE_ADDED`; with this, it is not possible to easily determine which is the latest. The MediaStore cursor parser solves this issue.

It should also be noted that right before and after the `query()`, two calls happen, respectively `ProviderHook.deactivate()` and `ProviderHook.activate()`.

Without these calls, the `query()` operation would trigger the proxied query. In subsection 6.2.4, we discussed the importance of having at all times consistency between the files stored on the MediaStore and the entries on the virtual ownership table. During the `finalizeInsert()` call, we do not have consistency, because bringing consistency is the actual purpose of that routine. Because of this, it is crucial to deactivate the proxied method and call the base original implementation. This is not easily doable, as the proxy is instantiated within a context which is not straightforward in access. The final solution was to place a flag variable, `STATE` in class `MediaProviderHook`, that can be set and reset by `activate()` and `deactivate()`. The thread safety is granted by the `synchronized` function.

**Query.** While insertions do not enforce any permissions, queries require them as an option. To recall, the appropriate permission is required in order to access to files that are owned by other apps. To allow this feature to be implemented, in conjunction with the virtual ownership table, it was used the approach of injecting instructions onto the SQL statement. The Pproxied query is shown in Listing 34.

```
public synchronized Cursor query(MethodBox methodBox, Uri url, String[] projection,
String selection, String[] selectionArgs, String sortOrder, Bundle originQueryArgs)
throws InvocationTargetException {
    Object[] updated = enforceQuery(url, selection, selectionArgs, originQueryArgs);
    selection = (String) updated[0];
    selectionArgs = (String[]) updated[1];

    Cursor cursor = super.query(methodBox, url, projection, selection, selectionArgs,
sortOrder, originQueryArgs);
    return new QueryRedirectCursor(cursor, COLUMN_NAME);
}
```

Listing 34 — Method `query()` in `MediaProviderHook`.

The injection happens on the first line of the method. The method `enforceQuery()`, shown in Listing 35, allows for checking if the appropriate permission is granted. If the permission is granted, no further action is required, and MediaStore will access to the full database. If the permission is not granted, the query must be injected with ID checks.

The method accepts a `Uri`, that represents the URI of the `ContentProvider` to access. Each media type in MediaStore has a respective database and thus URI. Then there are three parameters that contain the value of the low-level SQL query. Parameters `selection` and `selectionArgs` are self-explanatory, as they contain the selection statement and the arguments. Parameter `queryArgs` is just an encapsulation introduced in more recent Android

versions of those two variables into a `Bundle`. The duplication of selection arguments is done to keep the legacy method still working.

```
private Object[] enforceQuery(Uri uri, String selection, String[] selectionArgs,
Bundle queryArgs) {
    String permission = // Determine correct permission based on Uri
    int permissionState = // Get permission granted state

    final int uid = VBinder.getCallingUid(); // Virtual UID of the plugin app

    if (permission == null || permissionState == PackageManager.PERMISSION_DENIED) {
        // Permission is not granted! Need to only query for owned files!

        Integer[] ids = VMediaProvider.get().getIds(uid, uri);

        if(ids == null) { // I do not own any files: Select nothing
            selection = "1=0";
            selectionArgs = null;
        }
        else { // Select only the owned files
            if(selection == null || selection.isEmpty()) selection = "1=1";
            selection += " AND " + BaseColumns._ID + " IN (";
            for(int id : ids) {
                selection += " ? ,";
            }
            selection = selection.substring(0, selection.length() - 1);
            selection += " )";

            if (selectionArgs == null) selectionArgs = new String[0];
            String[] temp = Arrays.copyOf(selectionArgs, selectionArgs.length +
                ids.length);

            for (int i = 0; i < ids.length; i++)
                temp[selectionArgs.length + i] = String.valueOf(ids[i]);
            selectionArgs = temp;
        }

        queryArgs.putString(QUERY_ARG_SQL_SELECTION, selection);
        queryArgs.putStringArray(QUERY_ARG_SQL_SELECTION_ARGS, selectionArgs);
    }
    return new Object[]{selection, selectionArgs};
}
```

Listing 35 — Method `enforceQuery()` in `MediaProviderHook`.

The `enforceQuery` injection method first checks which is the correct permission for each type of file.

- `MediaStore.Video.Media.EXTERNAL_CONTENT_URI` → `READ_MEDIA_VIDEO`;
- `MediaStore.Images.Media.EXTERNAL_CONTENT_URI` → `READ_MEDIA_IMAGES`;
- `MediaStore.Audio.Media.EXTERNAL_CONTENT_URI` → `READ_MEDIA_AUDIO`;

The method then checks whether the permission has granted state. If the permission is granted, no injection should take place and thus `selection` and `selectionArgs` should remain unchanged.

Elsewhere, we get the current IDs for files owned by the plugin app under execution. If the IDs array is `null`, meaning we do not have any owned files, completely overwrite the selection statement with `1=0`. This statement is equivalent for `false`, and will not return anything.

Elsewhere, we modify both the selection and the respective arguments by following this rule:

```
selection: inject AND _id IN ( ? ) at the end of the selection
selectionArgs: inject the full list of owned IDs.
```

Following a slightly modified example as above, let's simulate a query authored by VUID 10002. Note that this example, for the explanation sake, has a notation abuse. The `VUID` column is not part of the MediaStore database, it is indeed part of the virtual ownership table. In Table 8, MediaStore and virtual ownership tables have been joined for clarity.

_ID	DISPLAY_NAME	RELATIVE_PATH	OWNER_PACKAGE_NAME	VUID
1	file.mp3	storage/emulated/0/Music/a	io.va.exposed64	10002
2	file.mp3	storage/emulated/0/Music/b	io.va.exposed64	10003
3	file.mp3	storage/emulated/0/Music/c	io.va.exposed64	10002

Table 8 — MediaStore table joined with virtual ownership table.

Before the injection, the statement is:

```
selection: _display_name = ?
selectionArgs: [file.mp3]
```

After the injection, the statement becomes:

```
selection: _display_name = ? AND _id IN (?, ?)
selectionArgs: [file.mp3, 1, 3]
```

By doing so, the file with `ID = 2`, that is not owned by VUID 10002, is excluded from the query.

Let's now assume a malicious app `com.example.malicious` with VUID 10004 tries to query for all files. Note that parameter `null` selects all files.

```
selection: null
selectionArgs: [null]
```

After the injection, the statement becomes:

```
selection: 1=0
selectionArgs: [null]
```

selecting effectively zero files.

**Delete.** A similar approach is used on the `delete()` method. When files are deleted from the MediaStore, a similar routine than the one on the `MediaProviderService` is launched to clean the virtual ownership table from leftover files. The only difference is that in this approach, timeliness is fundamental. We may have very fast looped deletion from the database, and the asynchronous solution offered by the Service is not valid anymore. A synchronous method is required, and to bypass the hook, the same implementation of `ProviderHook.deactivate()` was used.

### 6.2.6 MEDIASTORE ENTRY

The `MediaStoreEntry` class is a wrapper class to represent an entry on the MediaStore. This is used in conjunction with the `MediaStoreCursorParser` to cope with the limitation of the time being in seconds in SQL databases. `MediaStoreEntry` implements `Comparable`, which is fundamental to allow comparisons for sorting entries by time.

The columns are stored in a `HashMap` and the class is paired with a getter and a setter method for convenient access, as shown in Listing 36.

```
public void addColumn(final String name, final String value) {
    columns.put(name, value);
}

public String get(final String name) {
    return columns.get(name);
}
```

Listing 36 — Getter and setter in `MediaStoreEntry`.

```
public int compareTo(MediaStoreEntry o) {
    int thisId = Integer.parseInt(get(BaseColumns._ID));
    int thatId = Integer.parseInt(o.get(BaseColumns._ID));

    String thisDateStr = get(MediaStore.MediaColumns.DATE_ADDED);
    String thatDateStr = o.get(MediaStore.MediaColumns.DATE_ADDED);
    if(thisDateStr != null && thatDateStr != null) {
        int thisDate = Integer.parseInt(thisDateStr);
        int thatDate = Integer.parseInt(thatDateStr);

        // If dates are equal, compare by ID (IDs are generated sequentially)
        if(thisDate == thatDate) return Integer.compare(thisId, thatId);

        // Prefer comparison by date, if date column exists
        return Integer.compare(thisDate, thatDate);
    }

    // As a last resort, compare by ID
    return Integer.compare(thisId, thatId);
}
```

Listing 37 — Method `compareTo()` in `MediaStoreEntry`.

The core logic resides on the `compareTo()` method, shown in Listing 37. For a comparison between two `MediaStoreEntry` objects, the implementation gives priority to the column date. This is however possible only if the date is different. Being the date expressed in seconds, it is common in the real world for many entries to share the same date. In such case, the custom implementation checks and compares for the ID. IDs are sequentially generated by the system, so a greater number means newer file. With this approach it is possible to discriminate and infer the chronological order in which files have been added to the `MediaStore`.

### 6.2.7 MEDIASTORE CURSOR PARSER

The `MediaStoreCursorParser` class contains utility methods related to cursors. The method `parseCursorAndSort`, shown in Listing 38, is a utility method used to retrieve the entries from a cursor and return a sorted representation of them by date.

```
public static MediaStoreEntry[] parseCursorAndSort(Cursor cursor) {
    Vector<MediaStoreEntry> vector = new Vector<>();
    if (cursor != null) {
        while (cursor.moveToNext()) {
            int id;
            try {
                id = cursor.getInt(cursor.getColumnIndexOrThrow(BaseColumns._ID));
            } catch (IllegalArgumentException e) {
                throw new RuntimeException("Column " + BaseColumns._ID +
                    " not available on this cursor!");
            }

            String date = "";
            try {
                date = cursor.getString(cursor.getColumnIndexOrThrow(
                    MediaStore.MediaColumns.DATE_ADDED)
                );
            } catch (IllegalArgumentException e) {
                throw new RuntimeException("Column " + MediaStore.MediaColumns.DATE_ADDED +
                    " not available on this cursor!");
            }

            MediaStoreEntry entry = new MediaStoreEntry(id);
            entry.addColumn(MediaStore.MediaColumns.DATE_ADDED, date);
            vector.add(entry);
        }
        cursor.close();
    }
    Collections.sort(vector, Collections.reverseOrder());
    return vector.toArray(new MediaStoreEntry[0]);
}
```

Listing 38 — Method `parseCursorAndSort()` in `MediaStoreEntry`.

The method gets the columns from the cursor in a `while` loop. Every entry is safely checked to avoid missing columns. The list is then sorted by the sorting algorithm `Collections.sort()`. A sorted array of `MediaStoreEntry` objects is then returned.

### 6.3 CONFLICT AND FIX

These two implementations work perfectly if applied alone. However, when enabled at once, a clear and impactful conflict happens.

The direct file access countermeasure allows an app to only access to its designated directories – internal and external storage. This means that with the countermeasure active, any access to MediaStore `.json` files is strictly forbidden. This behavior is correct, as it allows keeping the private space of the container isolated from plugin apps.

When enabling the second countermeasure though, we see a conflict. All hooked methods run on the same process of the plugin app. In a sense, the framework cannot make any distinction whether the code that is being executed comes from the `.dex` of the plugin app, or from the hooked methods.

When accessing in both read and write mode to the MediaStore, we may need access to MediaStore `.json` files. To be precise, whenever a virtual application is launched, all caches are restored, meaning that at least one read to all three `.json` files is forced. Here we see the issue: since the logic that handles the virtual ownership is located obviously within the proxied method, that code is executed within the same process that runs the plugin app. It results that whenever the MediaStore countermeasure tries to access any of the `.json` files, the direct file access countermeasure blocks the operation, leading to an unusable framework.

That was not the only feature that suffered from the same cause. Also, the virtual permissioning implementation had similar results. That implementation needed a configuration file stored on the same directory to the `.json` files. Any time the virtual model tried to access to the `permissions.xml` file, the framework itself would block the access, making the entire framework useless.

It is not possible to discriminate in any way by whom the file access is done – the container or the plugin app. A novel approach must be implemented.

Clearly, this problem arises the main weakness of *security patching* existing software, in contrast with the stronger *secure by design* approach. A detailed analysis will be conducted in section 8.1.

### 6.3.1 INVOCATION STUB MANAGER

InvocationStubManager is a class that allows for proxies to be instantiated. It is responsible for injecting both proxied methods and proxied Services.

```
private void injectInternal() {
    if (VirtualCore.get().isMainProcess()) {
        return;
    }
    if (VirtualCore.get().isServerProcess()) {
        addInjector(new ActivityManagerStub());
        addInjector(new PackageManagerStub());
        return;
    }
    if (VirtualCore.get().isVAppProcess()) {
        addInjector(new LibCoreStub());
        addInjector(new ActivityManagerStub());
        addInjector(new PackageManagerStub());
        if (Build.VERSION.SDK_INT >= 28) {
            addInjector(new TransactionHandlerStub());
        }
        addInjector(HCallbackStub.getDefault());
        addInjector(new ISmsStub());
        addInjector(new ISubStub());
        addInjector(new DropBoxManagerStub());
        addInjector(new NotificationManagerStub());
        // ...
    }
}
```

Listing 39 — Method injectInternal() in InvocationStubManager is responsible for injecting proxies and system services.

All proxies are injected in batch, from the injectInternal() method, shown in Listing 39. There are methods to completely remove and inject an entire injector, but this poses a problem in our case. Entirely removing all MethodProxies within the same injector may void some of the functionality of the virtualization framework.

The implementation proposed in Listing 40 and Listing 41 allows for only removing and injecting a specific MethodProxy, making the entire process more controllable and secure.

```
public static <T extends IInjector, H extends MethodInvocationStub> boolean
    removeMethodProxy(Class<T> injectorClass, final String methodName) {

    H methodInvocationStub =
        InvocationStubManager.getInstance().getInvocationStub(injectorClass);

    if(methodInvocationStub == null) return false;

    return methodInvocationStub.removeMethodProxy(methodName) != null;
}
```

Listing 40 — removeMethodProxy() to selectively remove methodName from injectorClass. All other MethodProxies are left in operation.

```
public static <T extends IInjector, H extends MethodInvocationStub> boolean
    insertMethodProxy(Class<T> injectorClass, MethodProxy proxy) {

    H methodInvocationStub =
        InvocationStubManager.getInstance().getInvocationStub(injectorClass);
    if(methodInvocationStub == null) return false;

    return methodInvocationStub.addMethodProxy(proxy) != null;
}
```

Listing 41 — insertMethodProxy() to selectively add proxy to injectorClass.

### 6.3.2 LOCKED OPERATION

Alberto, in his implementation, created a class named `LockedOperation`. This class ensures that read and write operations on a file are done by acquiring a shared/exclusive lock for the entire duration of the operation.

Modifications were conducted as shown in Listing 42, by calling `prepare()` and `restore()` before and after each block of code that accesses a file.

```
public final <T> T read(final Supplier<T> operation) {
    try {
        prepare();

        return locker.performWithSharedLockOn(file, channel -> {
            ensureLoaded(channel);
            return operation.get();
        });
    } finally {
        restore();
    }
}
```

Listing 42 — Method read() in LockedOperation.

With this approach, it is possible to remove and restore selectively the `Os.open()` proxy, before and after each file read.

```
private synchronized void prepare() {
    InvocationStubManager.removeMethodProxy(LibCoreStub.class, "open");
}

private synchronized void restore() {
    InvocationStubManager.insertMethodProxy(
        LibCoreStub.class,
        new LibCoreStub.Open()
    );
}
```

Listing 43 — Methods `prepare()` and `restore()` in `LockedOperation`.

With the fix showed in Listing 43, every file access done through `LockedOperation` is exempt from any check on the path validity, because the source is trusted. During the file access, the original implementation of `os.open()` is called. Just after the access is complete, the proxy is inserted back, meaning that any subsequent file access will be checked for validity. Evaluation will be conducted in subsection 7.3.3.

# 7

## EVALUATION

The evaluation of the custom implementation has been carried out in three phases.

The first phase aimed at assessing whether the attacks could still be carried on. The evaluation was conducted by using the test apps created to show the vulnerabilities: FileTestRed and FileTestBlue. The main question to answer is: *does the implementation offer the same security performance compared to the original Android sandbox?*

The second phase aimed at checking whether the implementation was interfering with legitimate use case. Tests were conducted on six real world apps. The main question to answer is: *does the implementation offer reliability with legitimate file accesses?*

The third and last phase aimed at assuring that no other vulnerabilities have been introduced to the framework. The main question to answer is: *does the implementation introduce new vulnerabilities?*

This three-phase approach grants the effectiveness of the solution. For instance, let's assume a naive implementation that always blocks any file access, regardless of the legitimacy. This implementation would certainly pass phase one of the evaluation, but would not pass phase two. On the contrary, an implementation that always permits any file access would pass phase two but would not pass phase one.

## 7.1 PHASE ONE – TEST ON MALICIOUS APPS

The application I previously created for test purposes are also perfect for checking whether the vulnerabilities are still exploitable after the implementation. To recall, I have two identical apps with different package names: `com.example.filetestred` and `com.example.filetestblue`. They both offer a basic file manager and a MediaStore test.

### 7.1.1 DIRECT FILE ACCESS

In order to assess the validity of the sandbox, it is necessary to just play around with the file manager.

When FileTestBlue tries to access the private files from `com.example.filetestred`, an exception is thrown. This results in the denial of the file access. The behavior is documented on Figure 12.

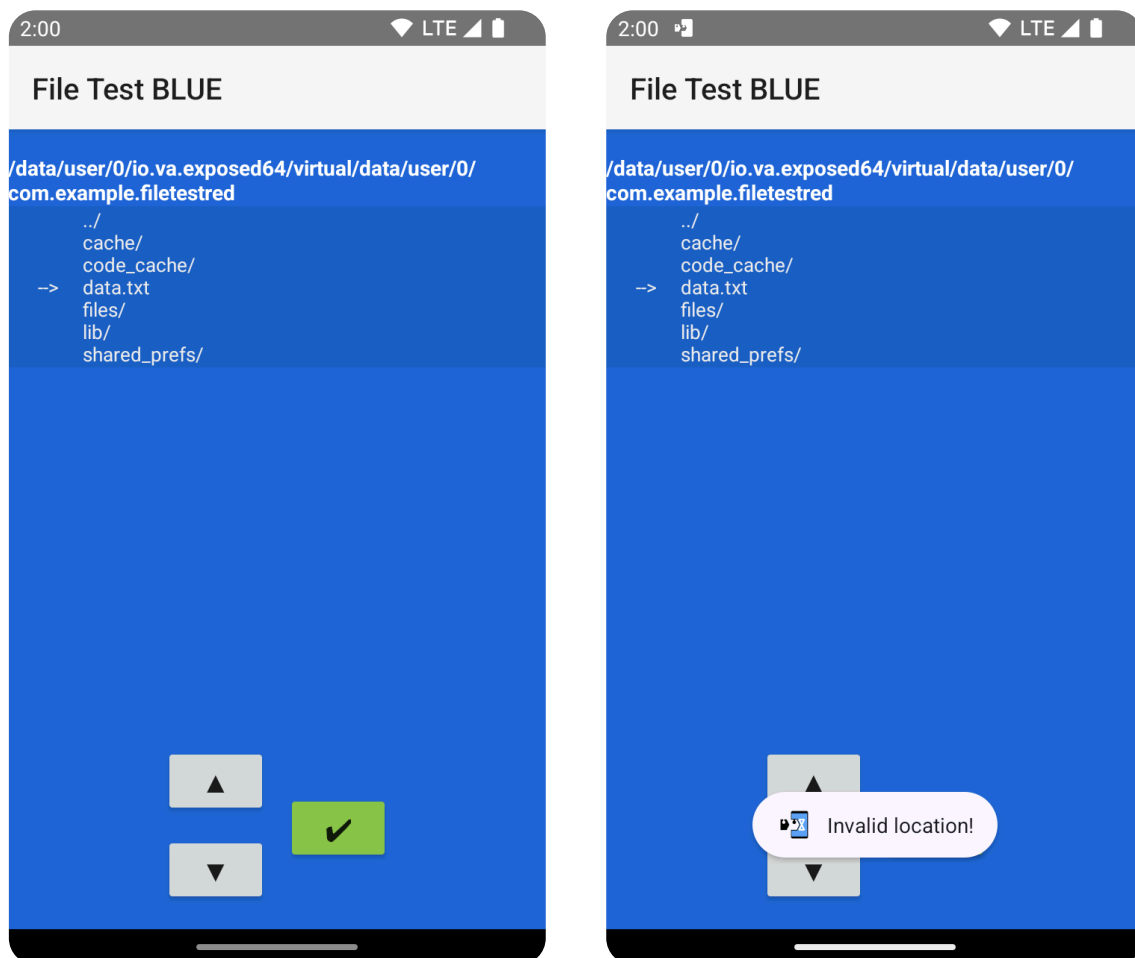


Figure 12 – Evaluation on direct file access, targeted to other app’s data.

## 7. EVALUATION

---

The exception is then propagated and captured by the FileTestBlue app and the Toast is displayed.

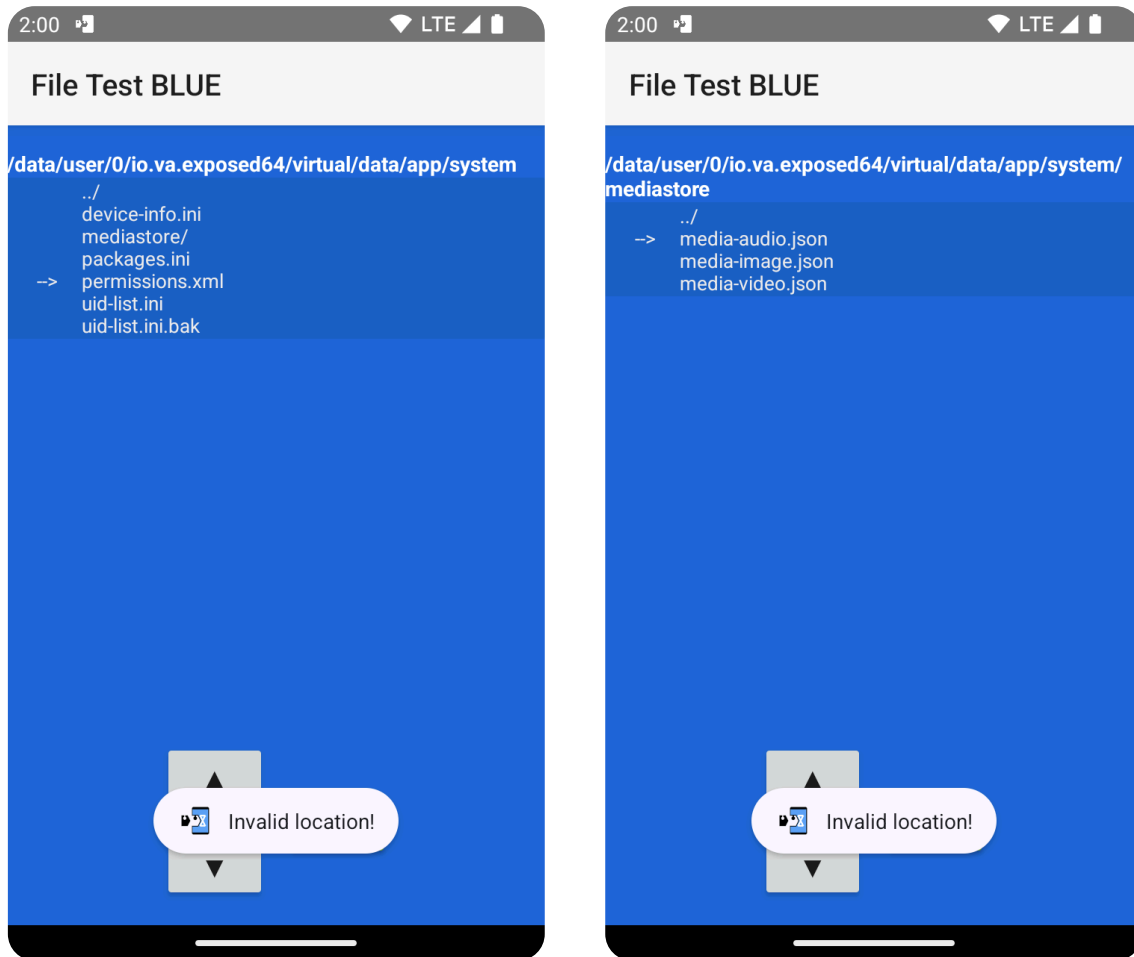


Figure 13 – Evaluation on direct file access, targeted to framework configuration files.

The same behavior is documented in Figure 13, when trying to access to `permissions.xml` or any MediaStore-related `.json`.

Inside the container's private directory, in fact, only the installation folder and the virtual private directory of the plugin app under execution are accessible. Any other file outside these locations is not accessible.

### 7.1.2 MEDIASTORE

The MediaStore section within the FileTest apps, that was previously used to show a basic attack, is also perfect for determining the performance.

Figure 14 shows that a newly created BLUE.mp3 file is perfectly visible within FileTestBlue, but is not within FileTestRed in absence of the right permission.

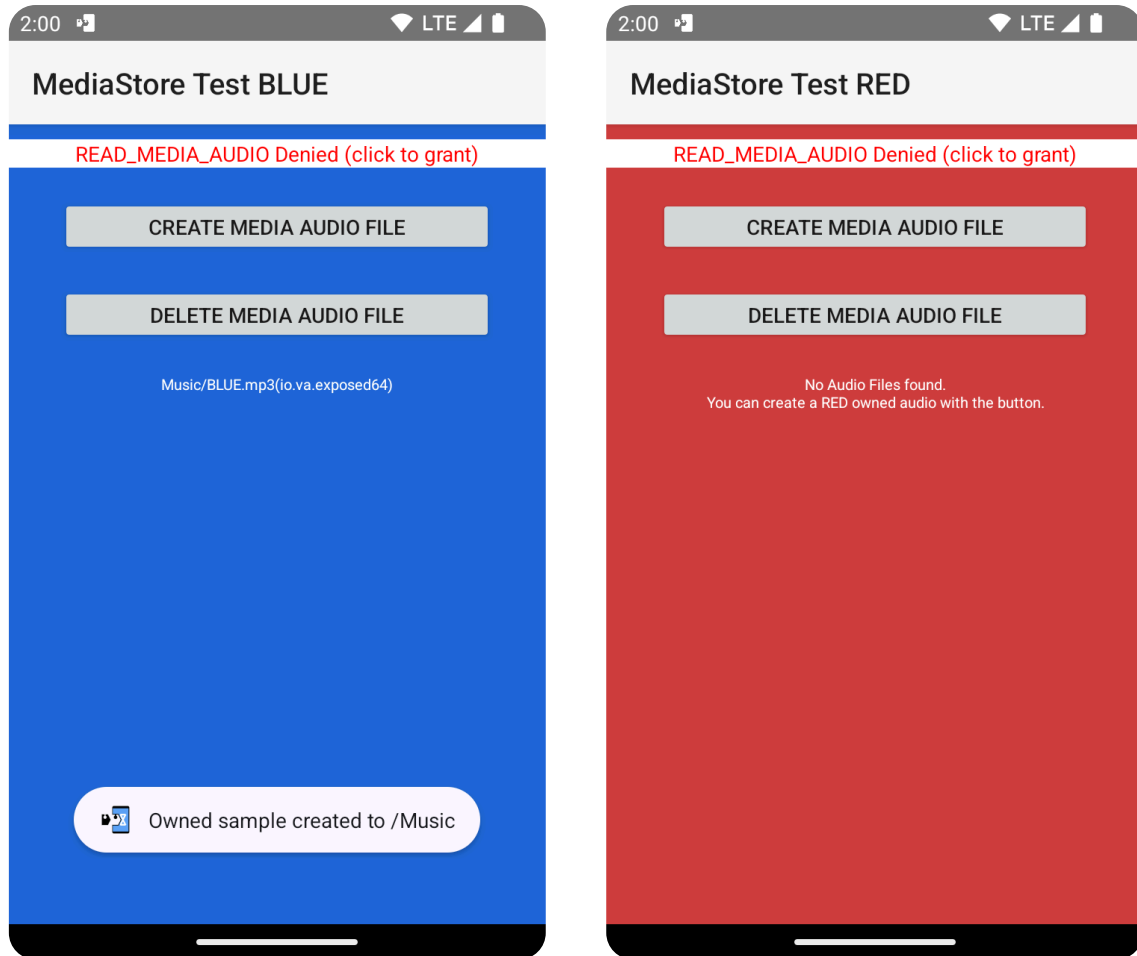


Figure 14 — Evaluation on MediaStore.

When the user grants `READ_MEDIA_AUDIO`, FileTestRed is capable of querying also for FileTestBlue files. Moreover, if I try to remove every file from the MediaStore, we do not have access to remove any other files that are owned by other plugin apps.

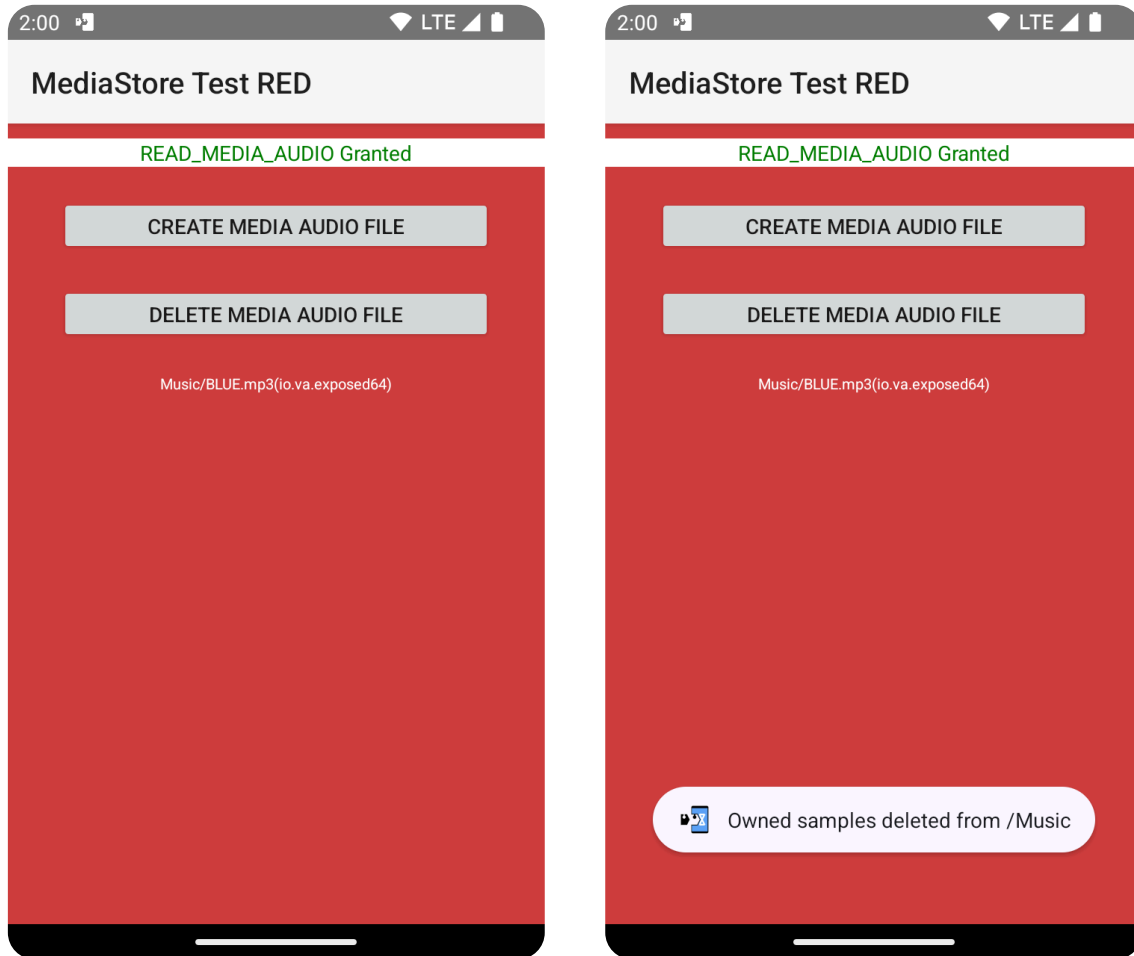


Figure 15 – Evaluation on MediaStore.

This surely shows that the two implementations offer the same experience and security level of the native Android sandbox.

### 7.2 PHASE TWO – TEST ON LEGITIMATE USE CASE

A selection of six simple real world apps has been utilized to assess the performance of the implementation. The goal was to play around with the apps, especially the features that notably require file or MediaStore access.

The following apps were used for testing:

- Open Camera [27]
- Food-Tracker [28]
- Capy Reader [29]
- Notes [30]
- GeoGebra [31]
- IPTV [32]

Unfortunately, the VirtualApp framework is open source only up to Android 8. With newer Android versions, system services and internal calls drastically changed. That is why intensive updates to the framework are required after each new Android version. VirtualXposed framework was successfully updated with unofficial support up to Android 15, but some features are still broken and prone to crash.

One notable example is the Autofill Service, which is responsible for automatically filling in text fields with user’s saved data. In the case of Android 14, there is an internal call from a `null` object, that leads to frequent crashes in presence of a `EditText` with the autofill flag set to `true`. In order to circumvent this issue, I had to disable the `MethodProxy` relative to that call. Adapting the framework to be fully compatible for Android 14 and newer versions is beyond the scope of this work.

However, in all the thorough tests carried out, no crashes attributable to the implementations were found. Intensive checks on the logcat confirmed this behavior, as no `FileNotFoundException` was ever recorded.

The only denied file access that is possible to see when running plugin apps is the `AppCenter.xml` file.

```
File access DENIED: /data/user/0/io.va.exposed64/shared_prefs/AppCenter.xml
```

The choice to disable such access has been discussed in *Microsoft AppCenter* subsection, page 33.

Listing 44 shows a sample of how the output of the logcat looks like when running GeoGebra.

```
File access GRANTED: /data/misc/keychain/pubkey_blacklist.txt
File access GRANTED: /data/misc/keychain/serial_blacklist.txt
File access GRANTED: /apex/com.android.conscrypt/cacerts/f013ecaf.0
File access GRANTED: /apex/com.android.conscrypt/cacerts/c90bc37d.0
File access GRANTED: /data/user/0/io.va.exposed64/virtual/data/user/0/
org.geogebra.android/files/generatefid.lock
File access GRANTED: /data/user/0/io.va.exposed64/virtual/data/user/0/
org.geogebra.android/shared_prefs/com.google.android.gms.
measurement.prefs.xml
```

Listing 44 — The output from the logcat shows which files are accessed.

### **7.3 PHASE THREE – CHECK FOR VULNERABILITIES**

Some of the approaches used on the implementation may arise some concern about the overall security. In general, fixing a vulnerability is successful only if the implementation does not introduce others.

I analyzed all the possible weaknesses about my implementation and designed a custom app for each of them in order to assess the performance.

#### **7.3.1 TIME COST**

Any logic that is added to a base implementation adds computational complexity, which in turn is often noticed in an increased time of execution. It is expected for the base framework alone to add complexity, as each proxied call is surely slower than what we experience in a native non-proxied APK execution. By comparing these two complexities with the complexity of the proposed implementation, we may get interesting insights.

In order to address the time cost of the implementation, I coded a basic app that loops 100 file and MediaStore accesses. Time is measured to complete all the 100 operations.

#### **DIRECT FILE ACCESS**

In order to better understand the negative effect on time we should first evaluate the performance in a clean native execution. The first run is done on the installed APK, so no virtualization framework is used.

## 7. EVALUATION

---

<b>Run</b>	<b>Time [ms]</b>	<b>Time per Insert [ms]</b>
1	101	1.01
2	112	1.12
3	102	1.02
4	116	1.16
5	127	1.27
6	97	0.97
7	117	1.17
8	122	1.22
9	103	1.03
10	122	1.22
<b>AVG</b>	<b>112</b>	<b>1.12</b>

Table 9 – Time cost for 100 file reads, running as a native installed APK.

As shown in Table 9, in average, a full file access takes 1.12 ms.

<b>Run</b>	<b>Time [ms]</b>	<b>Time per Insert [ms]</b>
1	128	1.28
2	145	1.45
3	202	2.02
4	127	1.27
5	121	1.21
6	122	1.22
7	165	1.65
8	116	1.16
9	116	1.16
10	117	1.17
<b>AVG</b>	<b>136</b>	<b>1.36</b>

Table 10 – Time cost for 100 file reads, running on the virtualization framework without the secure implementation.

As seen on Table 10, the overhead caused by the virtualization framework is negligible. Averaging 1.36 ms, the difference is truly marginal.

Run	Cached		Default	
	Time [ms]	Time per Insert [ms]	Time [ms]	Time per Insert [ms]
1	126	1.26	136	1.36
2	138	1.38	141	1.41
3	147	1.47	144	1.44
4	121	1.21	133	1.33
5	120	1.2	133	1.33
6	127	1.27	136	1.36
7	120	1.2	135	1.35
8	127	1.27	132	1.32
9	165	1.65	143	1.43
10	152	1.52	137	1.37
<b>AVG</b>	<b>134</b>	<b>1.34</b>	<b>137</b>	<b>1.37</b>

Table 11 — Time cost for 100 file reads, running on the virtualization framework with the secure implementation.

Table 11 shows the results of the run. Two tests were done on the custom implementation: one with the cache enabled and a second with the cache disabled. This is because the cache effectively bypasses the check on the path, making the whole complexity lower. Cached is slightly faster than not cached. However, the most surprising discovery is that these two times are actually comparable to the framework without the secure implementation.

This is a sign that the implementation itself is very lightweight and does not impact at all the execution. The small difference in the average is just a factor of the randomness of the tests.

**MEDIASTORE**

A similar test is conducted on the MediaStore access. 100 looped insertions are computed, and the total time is calculated. In theory this evaluation should produce bigger differences than what we've seen on the direct file access.

<b>Run</b>	<b>Time [ms]</b>	<b>Time per Insert [ms]</b>
1	2440	24.4
2	2342	23.42
3	2646	26.46
4	2406	24.06
5	2464	24.64
6	2306	23.06
7	2327	23.27
8	2709	27.09
9	2175	21.75
10	2709	27.09
<b>AVG</b>	<b>2452</b>	<b>24.52</b>

Table 12 — Time cost for 100 MediaStore file insertions, running as a native installed APK.

As reported in Table 12, the 100 executions were concluded in average in 2452 ms, with a rather stable performance across all ten runs. We immediately see that each MediaStore insertion has a rather large overhead, averaging 24.52 ms.

Then, other ten runs were done on the base virtualization framework, without any custom implementation.

<b>Run</b>	<b>Time [ms]</b>	<b>Time per Insert [ms]</b>
1	2803	28.03
2	2383	23.83
3	3578	35.78
4	2468	24.68
5	2522	25.22
6	2288	22.88
7	2373	23.73
8	2329	23.29
9	2458	24.58
10	2807	28.07
<b>AVG</b>	<b>2601</b>	<b>26.01</b>

Table 13 — Time cost for 100 MediaStore file insertions, running on the virtualization framework without the secure implementation.

Table 13 shows that in average, the 100 insertions were completed in 2601 ms, surprisingly close to the native execution. The virtualization framework does a good job at optimizing the resources. Although some overhead is expected, 1.5 ms added for each single insertion seems like a good result.

As a last test, ten runs with 100 insertions with all implementations active.

Run	Time [ms]	Time per Insert [ms]
1	7504	75.04
2	7227	72.27
3	7214	72.14
4	6629	66.29
5	3849	38.49
6	5804	58.04
7	5392	53.92
8	4938	49.38
9	5427	54.27
10	6005	60.05
<b>AVG</b>	<b>5999</b>	<b>59.99</b>

Table 14 — Time cost for 100 MediaStore file insertions, running on the virtualization framework with the secure implementation.

Table 14 documents that the implementation increases by a good margin the computation time. Averaging roughly 60 ms per insertion, we see a doubling in time compared to the base framework. This is easily explained by the logic of the implementation. Each insertion requires the back-end to also perform a query. This means that a single MediaStore access hides another access. This roughly results in doubling the access time.

When dealing with security, we should expect a decrease of performance. Each check is computed makes the framework more secure, but also trades some computational time.

### 7.3.2 FAST MEDIASTORE OPERATIONS

For the MediaStore implementation, the previous test proved to slow down the computation time by a factor of two. This may not be a problem for a single query, but might be a problem for looped sequential access.

The previous test showed that 100 sequential insertion are possible, but did not prove that all these insertions were correctly parsed and populated on the virtual ownership table.

In order to evaluate if the table is still consistent when performing fast insertions, we have to check if the IDs are correct.

After the 100 insertions, and the `.json` file looked like Listing 45.

```
[{"id":100003546,"owner":10002}, {"id":100003547,"owner":10002},
{"id":100003544,"owner":10002}, {"id":100003545,"owner":10002},
{"id":100003550,"owner":10002}, {"id":100003551,"owner":10002},
{"id":100003548,"owner":10002}, {"id":100003549,"owner":10002},
{"id":100003538,"owner":10002}, {"id":100003539,"owner":10002},
{"id":100003536,"owner":10002}, {"id":100003537,"owner":10002},
{"id":100003542,"owner":10002}, {"id":100003543,"owner":10002},
{"id":100003540,"owner":10002}, {"id":100003541,"owner":10002},
{"id":100003530,"owner":10002}, {"id":100003531,"owner":10002},
{"id":100003528,"owner":10002}
...
```

Listing 45 — File `media-audio.json` after 100 insertions.

The `.json` file actually holds 100 lines, so no file is missing. To double-check, it was extracted every `"id"` field from the file, and sorted the list. The outcome is the list of progressive IDs generated by the OS. No IDs are missing, and the number is precisely 100.

This proves, despite having doubled the access time, that everything is still consistent.

### 7.3.3 METHODPROXY TEMPORARY DISABLING

In section 6.3, it has been discussed that in order to properly access to `MediaStore` during the update of the virtual ownership table, it is necessary to briefly disable and re-enable the `Os.open()` `MethodProxy`.

This might be exploited by a malicious app, that constantly loops file accesses to a protected resource. The idea is that when the `MethodProxy` is briefly removed, the malicious app might be able to complete the access.

An app with a `Service` that runs in the background was created. The `Service`, depicted in Listing 46, has a logic that spawns a new thread, and infinitely tries accessing to `permissions.xml`. This file is protected by the sandbox implementation and thus all the accesses fail in normal conditions.

```
new Thread(() -> {
    try {
        File file = new File(getFilesDir(), "sample.txt");
        BufferedReader reader = new BufferedReader(new FileReader(file))

        String line;
        while ((line = reader.readLine()) != null) {
            Log.d(TAG, line);
        }

        reader.close();
    } catch (IOException ignore) {}
}).start();
```

Listing 46 — Code used to loop file accesses.

Another plugin app was then opened. This triggers file accesses and permission checks, with the effect of disabling and re-enabling the proxy every time. In no circumstance the malicious app was able to get access to the protected file.

This is explained by the fact that plugin apps run in separate processes, with separate sets of proxies. Disabling a proxy in one process, does not disable proxies in others. This is confirmed by another ad-hoc test, where it was selectively disabled a proxy for one specific package name, and effectively all other plugin apps with different package name still had the proxy enabled.

# 8

## CONCLUSION

This work shows that it is possible to reinstate the behavior of the sandbox by building a model on top of the virtualization framework. The end user experiences no difference from the installed APKs and the attack surface is truly not applicable anymore. The implementation, in fact, proved to be successful on mitigating the sandbox related attacks to VirtualApp.

The performance is optimal for direct file access, and acceptable for MediaStore operations, considering that any security-related check always requires a computational tradeoff.

This work is however just a proof of concept. App-level virtualization, if not designed properly, poses security issues on the sandbox, and a proper architecture should be implemented.

### 8.1 SECURE BY DESIGN

Next-gen app-level virtualization frameworks should follow the *secure by design* approach. With this paradigm, security is among the main aspects to consider when performing the first stages of software design.

However, in practice, especially within small to medium-sized enterprises, security is often deprioritized due to budget and time constraints. This choice may accelerate

the production phase but often results in systems that require costly security update revisions to address vulnerabilities.

Moreover, if a software is not designed to be secure from the ground up, patching may be complicated. I faced this issue when dealing with the disabling of MethodProxies. This approach is more like a workaround, and not a proper solution. It clearly works, as the evaluation showed, but it just adds more pieces to an already intricate framework.

When developing a new virtualization framework, a proper threat modeling phase should be conducted. Models like *STRIDE* are often considered to be the most effective methodologies for systematically identifying and categorizing potential security threats.

STRIDE is the acronym for:

- Spoofing
- Tampering
- Repudiation
- Information Disclosure
- Denial of Service
- Elevation of Privilege

The analysis on possible threats before the coding phase, usually done by playing the attacker's side, allows the team to understand and mitigate the attack surfaces from the ground up.

Attacks performed on the VirtualApp framework fall under Information Disclosure and Elevation of Privilege.

This experience highlights the difficulty of retrofitting security into an existing system and underscores the need for secure by design principles.

## 8.2 REAL WORLD FRAMEWORKS TESTED

These results should warn people that have been or are still using virtualization frameworks, as they might be exposed to severe vulnerabilities. Most of the frameworks are not open source, making it very difficult to get the hands on the code to perform tests.

VirtualApp is an old framework, not open source anymore. For these reasons, I decided to attempt the same sandbox attacks on some of the most downloaded and modern frameworks available on the Google Play Store to this date.

## 8. CONCLUSION

---

The attack was performed on:

- Dual Space, 100+ millions downloads [33]
- Parallel Space, 100+ millions downloads [34]
- Parallel App, 10+ millions downloads [35]
- Clone App, 5+ millions downloads [36]

I performed both the MediaStore and sandbox attack I designed for VirtualApp, and with my surprise, they are both successful. These frameworks have a combined total of 215+ million downloads, possibly exposing users to data leakage.

We should also consider that it is not necessary to install a malicious plugin app that steals data through the sandbox vulnerability. Data could also be stolen directly by the framework itself. In this case, there is little that can be done, as only the trustworthiness of the developer plays a role.

For example, let's assume a closed-source framework that enforces the sandbox with an equivalent implementation as the one proposed on this work. Being closed-source, we cannot check on the source code with ease, as usually reversing is required. Let's also consider that most closed-source enterprise-level code is obfuscated for obvious purposes. What we can do instead is test if the attacks succeed, and we assume they do not. This, however, tests the attack surface from the plugin apps. The attack surface might come directly from the framework itself. In fact, the framework has technically access to all app files, because they are stored on the virtual file system. The framework may enforce the sandbox for plugin apps, but might also steal data itself. This is the reason why virtualization frameworks, especially if closed-source, can become a severe threat for user's privacy.

On this matter, zLabs recently discovered a virtualization-based malware [37]. The discovered threat would collect data and passwords of unaware users, thanks to the proxying functionality of virtualization frameworks.

In conclusion, vulnerabilities are often not properly considered by end users, usually for the lack of proper education on the matter. Users often tend to minimize the true risk of being attacked, and this increases the chance for cyber-criminals to succeed.

# 9

## FUTURE WORK

### 9.1 VIRTUALPATCH

A work by Simeone Pizzi [38] showed that it is possible to apply security patches by exploiting dynamic hooking of methods. VirtualPatch is a fork of VirtualApp, that allows dynamic loading of custom-made patches.

VirtualPatch offers a library to define patches. Patches are intended to fix vulnerabilities that are usually updated through the Android monthly security patches. Some manufacturers are slow at updating their devices, possibly exposing hundreds of thousands of users to vulnerabilities. By looking at Android Security Bulletins, it is possible to check on the source code modifications. Then, thanks to the VirtualPatch library, it is possible to define hooks for the vulnerable methods. The implementation on hooks should match the fixed version of the method. This way, when the method is called, the hooked and fixed implementation is executed in place of the vulnerable one.

The three projects can be combined in an all-in-one framework, that allows for executing apps in a clean and controlled environment.

In particular, such framework would:

1. Comply with the Android sandbox;
2. Comply with the Android permissions;
3. Give the possibility to patch known vulnerabilities.

This would give the basis for the ultimate secure execution of Android apps, though some issues might slow the development down.

As discussed before, retrofitting is always a problem with medium-sized to large projects. It is in fact very easy to end up with conflicting components, that require even more fitting. Moreover, there are issues with hooking frameworks compatibilities.

VirtualPatch uses YAHFA [39], that is, similarly to Epic, a hooking framework for Android ART. It supports more architectures than Epic and unofficial support has been extended up to Android 12. However, development stopped in 2022, and no more updates are being published. The same happens for all major hooking frameworks available on the scene right now.

## 9.2 FULL ANDROID VIRTUALIZATION

In Android 16, Google added support for running a full Linux VM based on Debian. This is actually the first usable project of full OS virtualization in Android. Full virtualization, in contrast with application-level, does not break the Android sandbox, as the entire Android OS, including the kernel and user space, runs in an isolated virtual machine, preserving the integrity and separation of app processes and permissions.

Up to this day though, it is not allowed to load an arbitrary ISO, as users have to stick with the preconfigured default Debian environment. This project is very promising, as it could be the base for running full Android virtualization.

## REFERENCES

- [1] “Multiple Accounts Coming to WhatsApp.” [Online]. Available: <https://blog.whatsapp.com/multiple-accounts-coming-to-whatsapp?lang=en>
- [2] “WhatsApp beta for iOS 25.2.10.70: what's new.” [Online]. Available: <https://wabetainfo.com/whatsapp-beta-for-ios-25-2-10-70-whats-new>
- [3] *VirtualApp* – *GitHub*. [Online]. Available: <https://github.com/asLody/VirtualApp>
- [4] Deshun Dai, Ruixuan Li, Junwei Tang, Ali Davanian, and Heng Yin, “Parallel Space Traveling: A Security Analysis of App-Level Virtualization in Android,” in *SACMAT '20: The 25th ACM Symposium on Access Control Models and Technologies*, Jun. 2020. doi: <https://doi.org/10.1145/3381991.3395608>.
- [5] Alberto Lazari, “Towards Secure Virtual Apps: Bringing Android Permission Model to Application Virtualization,” 2024. doi: <https://hdl.handle.net/20.500.12608/80205>.
- [6] *Apktool* – *GitHub*. [Online]. Available: <https://github.com/iBotPeaches/Apktool>
- [7] “Shedun – Wikipedia.” [Online]. Available: [https://en.wikipedia.org/wiki/Shedun?utm\\_source=chatgpt.com](https://en.wikipedia.org/wiki/Shedun?utm_source=chatgpt.com)
- [8] “Trojanized Adware Family Abuses Accessibility Service.” [Online]. Available: <https://www.lookout.com/threat-intelligence/article/shedun-trojanized-adware>
- [9] “This is Doom running on Android 16's Linux Terminal.” [Online]. Available: <https://www.androidauthority.com/android-16-linux-terminal-doom-3521804/>
- [10] “Android Virtualization Framework (AVF) overview.” [Online]. Available: <https://source.android.com/docs/core/virtualization>
- [11] *VirtualXposed* – *GitHub*. [Online]. Available: <https://github.com/android-hacker/VirtualXposed>
- [12] *Epic* – *GitHub*. [Online]. Available: <https://github.com/tiann/epic>
- [13] Valerio Costamagna and Cong Zheng, “ARTDroid: A Virtual-Method Hooking Framework on Android ART Runtime,” in *Proceedings of the Workshop on Inno-*

- vations in Mobile Privacy and Security IMPS at ESSoS'16*, Apr. 2016. [Online]. Available: [https://ceur-ws.org/Vol-1575/paper\\_10.pdf](https://ceur-ws.org/Vol-1575/paper_10.pdf)
- [14] Marvin Wißfeld, “ArtHook: Callee-side method hook injection on the new Android runtime ART,” 2015. [Online]. Available: [https://publications.cispa.saarland/143/1/arthook\\_thesis.pdf](https://publications.cispa.saarland/143/1/arthook_thesis.pdf)
- [15] “YAHFA – Hook Framework in the ART environment.” [Online]. Available: <https://rk700.github.io/2017/03/30/YAHFA-introduction/>
- [16] Lei Zhang *et al.*, “App in the Middle: Demystify Application Virtualization in Android and its Security Threats,” in *Proceedings of the ACM on Measurement and Analysis of Computing Systems, Volume 3, Issue 1*, Mar. 2019. doi: <https://doi.org/10.1145/3322205.3311088>.
- [17] Tongbo Luo, Cong Zheng, Zhi Xu, and Xin Ouyang, “Anti-Plugin: Don't Let Your App Play As An Android Plugin.” [Online]. Available: <https://www.blackhat.com/docs/asia-17/materials/asia-17-Luo-Anti-Plugin-Don't-Let-Your-App-Play-As-An-Android-Plugin-wp.pdf>
- [18] Marco Alecci, Riccardo Cestaro, Mauro Conti, Ketan Kanishka, and Eleonora Losiouk, “Mascara: A Novel Attack Leveraging Android Virtualization,” Oct. 20, 2020. doi: <https://doi.org/10.48550/arXiv.2010.10639>.
- [19] Simone Zerbini, Samuele Doria, Primal Wijesekera, Serge Egelman, and Eleonora Losiouk, “Matrioska: A User-Centric Defense Against Virtualization-Based Repackaging Malware on Android,” in *2024 Annual Computer Security Applications Conference (ACSAC)*, Dec. 2024. doi: <https://doi.org/10.1109/ACSAC63791.2024.00073>.
- [20] Cong Zheng, Tongbo Luo, Zhi Xu, Wenjun Hu, and Xin Ouyang, “Android Plugin Becomes a Catastrophe to Android Ecosystem,” in *RESEC '18: Proceedings of the First Workshop on Radical and Experiential Security*, May 2018. doi: <https://doi.org/10.1145/3203422.3203425>.
- [21] Antonio Ruggia, Eleonora Losiouk, Luca Verdame, Mauro Conti, and Alessio Merlo, “Repack Me If You Can: An Anti-Repackaging Solution Based on Android Virtualization,” in *ACSAC '21: Proceedings of the 37th Annual Computer Security Applications Conference*, Dec. 2021. doi: <https://doi.org/10.1145/3485832.3488021>.
- [22] Luman Shi *et al.*, “VAHunt: Warding Off New Repackaged Android Malware in App-Virtualization's Clothing,” in *2024 Annual Computer Security Applications Conference (ACSAC)*, Nov. 2020. doi: <https://doi.org/10.1145/3372297.3423341>.
- [23] *FileInputStream.java – Android Code Search*. [Online]. Available: <https://cs.android.com/android/platform/superproject/+/android15-qpr2-release:libcore/ojuni/src/main/java/java/io/FileInputStream.java;l=179>

- [24] *IoBridge.java* – *Android Code Search*. [Online]. Available: <https://cs.android.com/android/platform/superproject/+/android15-qpr2-release:libcore/luni/src/main/java/libcore/io/IOBridge.java;l=560>
- [25] *ForwardingOs.java* – *Android Code Search*. [Online]. Available: <https://cs.android.com/android/platform/superproject/+/android15-qpr2-release:libcore/luni/src/main/java/libcore/io/ForwardingOs.java;l=563>
- [26] *Linux.java* – *Android Code Search*. [Online]. Available: <https://cs.android.com/android/platform/superproject/+/android15-qpr2-release:libcore/luni/src/main/java/libcore/io/Linux.java;l=172>
- [27] *Open Camera* – *F-Droid*. [Online]. Available: <https://f-droid.org/en/packages/net.sourceforge.opencamera/>
- [28] *Food-Tracker* – *F-Droid*. [Online]. Available: <https://f-droid.org/en/packages/org.secuso.privacyfriendlyfoodtracker/>
- [29] *Capy Reader* – *F-Droid*. [Online]. Available: <https://f-droid.org/en/packages/com.capyreader.app/>
- [30] *Notes* – *F-Droid*. [Online]. Available: <https://f-droid.org/it/packages/org.secuso.privacyfriendlynotes/>
- [31] *GeoGebra* – *APKMirror*. [Online]. Available: <https://www.apkmirror.com/apk/geogebra/geogebra-graphing-calculator/geogebra-graphing-calculator-5-2-879-0-release/>
- [32] *IPTV* – *APKMirror*. [Online]. Available: <https://www.apkmirror.com/apk/alexander-sofronov/iptv-2/>
- [33] *Dual Space – Multiple Accounts*. [Online]. Available: <https://play.google.com/store/apps/details?id=com.ludashi.dualspace>
- [34] *Parallel Space – Multiple Accounts*. [Online]. Available: <https://play.google.com/store/apps/details?id=com.lbe.parallel.intl>
- [35] *Parallel App – Dual App Cloner*. [Online]. Available: <https://play.google.com/store/apps/details?id=com.excean.parallelspace>
- [36] *Clone App – Parallel Dual Space*. [Online]. Available: <https://play.google.com/store/apps/details?id=com.pengyou.cloneapp>
- [37] “Your Mobile App, Their Playground: The Dark side of the Virtualization.” [Online]. Available: <https://zimperium.com/blog/your-mobile-app-their-playground-the-dark-side-of-the-virtualization>
- [38] Simeone Pizzi, “VirtualPatch: fixing Android security vulnerabilities with app-level virtualization,” 2022. doi: <https://hdl.handle.net/20.500.12608/32823>.

[39] [Online]. Available: <https://github.com/PAGalaxyLab/YAHFA>