Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea Magistrale in Informatica

VirtualPatch: fixing Android security vulnerabilities with app-level virtualization

Master's Degree Thesis

Relatore
Prof.ssa Eleonora Losiouk
Università di Padova

Controrelatore
Prof. Luca Verderame
Università di Genova

Laureando
Simeone Pizzi
1219375

Academic Year 2021-2022

"You're going to be all right. You just stumbled over a stone in the road. It means nothing. Your goal lies far beyond this. Doesn't it? I'm sure you'll overcome this. You'll walk again... soon."

Kentaro Miura

Abstract

Since its first release, the Android Operating System (OS) has been affected by a significant issue: the existence of multiple customized versions, handled by different mobile device vendors. One of the main consequences of the Android fragmentation issue regards the distribution of security updates to end-user devices. In particular, I have focused on the time required by Google and other mobile vendors to send security updates. I have found that, on average, Google takes more than 84 days to send an update, after its development is already complete, while Samsung takes, on average, over 39 days to integrate a Google security patch in its custom Android OS. During this time window, end-users are left exposed to attackers.

In this thesis, I propose VirtualPatch, a solution aimed at allowing the immediate distribution of Android security patches after their development, thus shrinking the aforementioned time window. VirtualPatch is a virtualization-based approach that protects apps by loading security patches targeting different Android architecture layers and, being executed at the application-layer, it does not require an update of the underlying Android OS. I chose seven Common Vulnerabilities and Exposures from the Android Security Bulletins and managed to successfully implement and deploy the associated security patches through my solution. Moreover, while the state-of-art already proved the runtime overhead introduced by the virtualization technique to be negligible, I measured also the average time required to load the security patches, which I found to be less than 60 milliseconds. Overall, VirtualPatch is an effective and efficient solution addressing the issue of the security patch distribution for Android users. Given the significance of the issue, I really hope to make a contribution to the whole Android community.

Contents

List of figures

List of tables

1 Introduction

In the last decade, mobile devices and in particular smartphones have found widespread utilization. A recent report [1] estimates the number of unique mobile device users to be around 6 billion and forecasts this number to keep increasing in the next few years. Among different mobile operating systems, Android is the undisputed leader, being installed on more than 71% of smartphones [2]. In comparison, the closest competitor is iOS, installed on about 28% of mobile devices, less than half of Android OS. There are many different vendors that produce Android smartphones, and it is probably thanks to this that Android devices have found such a widespread utilization. This is made possible by the Android Open Source Project (AOSP), which provides the source code of Android OS to the vendors. On the other hand, having many different Android device makers has also negative effects, such as fragmentation.

Fragmentation in Android is caused by the vendors customizing the stock Android OS, in order to make their devices look more appealing in the eyes of potential customers. This customization in turn makes it difficult to deliver software updates: when the source code of the AOSP is updated, the responsibility of publishing these updates and deploying them to user devices falls to the vendors themselves. Vendors need to adapt the code to their own customized versions of the OS, a process that requires time and effort, and, as a consequence, Android devices are not always updated, and when they are the updates are often delayed.

The Android fragmentation problem has led in the past to sporadic security updates, and for this reason, Google has worked to make it easier to deliver security updates to Android devices. The results of these efforts are *project Treble* [3], a refactor of the Android architecture that separates the OS framework from the device-specific low-level software, and *project Mainline* [4], which makes it possible to update some components of the Android Operating system without a full system update, through Google Play system updates. However, while a step in the right direction, these solutions only cover a small part of the Android architecture, and so system updates are still required for most of the security patches.

In order to guarantee users a certain level of security, the licensing agreement between Google and device manufacturers reportedly stipulates that they must provide security updates for at least two years after a device is released, and at least four security updates during the first year [5].

These terms are broad enough that the actual security patch situation changes from vendor to vendor, and from device to device. For instance, taking into consideration the 3 biggest Android device vendors, Samsung publishes monthly, quarterly, or biannual firmware security updates for different devices, and promises at least 4 years of security updates for devices launched in 2019 or later, without however specifying how regular the updates will be [6]; Xiaomi publishes monthly or quarterly security updates for different devices, and promises security updates for at least two years after launch [7]; Huawei publishes monthly or quarterly security updates for different devices, but does not mention for how long after launch these updates will be published [8]. In all cases, the vendors specify that the update schedule is subject to change, so it is not possible for users to know how regular the updates will be before buying a device.

With so many devices and so many users involved, security updates in Android are a very critical issue. In this thesis, I first try to determine the current security patch situation in Android, by analyzing data from official Android Security Bulletins as well as data about security updates for the biggest Android device vendor (Samsung). Then, I propose a potential solution that leverages app-level virtualization to deploy security patches to user devices, without requiring modifications of the underlying operating system or special privileges.

The rest of this thesis is organized as follows:

- in Chapter 2, I analyze the current situation of security updates in Android, first looking at the delay from when a security patch is committed to the AOSP repository to when it is published in an Android Security Bulletin (i.e. Section 2.1), and then at the delay incurred when the biggest Android vendor (Samsung) integrates a security patch into its customized version of the Android OS (i.e. Section 2.2);
- in Chapter 3, I give some background information about the Android architecture (i.e. Section 3.1), Virtualization (i.e. Section 3.2), and how an app-level virtualization Framework for Android works (i.e. Section 3.3);
- in Chapter 4, I provide an overview of previous work related to the analysis of Android Security Bulletins (i.e. Section 4.1), alternative ways to deploy security updates in Android (i.e. Section 4.2), and app-level virtualization in Android (i.e. Section 4.3);
- in Chapter 5, I describe first the design (i.e. Section 5.1) and then the implementation (i.e. Section 5.2) of VirtualPatch, a solution that uses app-level virtualization to deploy security updates on Android devices;
- in Chapter 6, I discuss the evaluation of VirtualPatch, done by implementing security patches for 7 real Common Vulnerabilities and Exposures (CVE)s (i.e. Section 6.1 through Section 6.4) and analyzing the time that it takes to load these patches (i.e. Section 6.5) in VirtualPatch;
- in Chapter 7, I discuss the results I obtained and the limitations of VirtualPatch, and give some ideas for future extensions of this work.

2 Motivation

In order to better understand the actual situation of security patches in Android, I scraped some data from security bulletins and from the official Samsung website [1] and analyzed it. In the following sections, I give an overview of the data collection process and report my findings on Android security updates.

2.1 Android Security Bulletin

Google publishes monthly Android Security Bulletins, which contain details about security vulnerabilities affecting Android devices. These details include a list of CVEs, with references to commits in the Android repositories that fixed those vulnerabilities. Since some of the Android components are not open-source, not all CVEs have references to the commits.

To each security bulletin, it corresponds a *security patch level*. Vulnerabilities reported on a certain monthly bulletin are fixed on devices that have an operating system version with a security patch level greater or equal to the bulletin date.

According to Google, "Android partners are notified of all issues at least a month before publication" [9] so that they have the time to adapt the patches to their customized version of Android and deploy security updates.

I scraped all Android Security Bulletins published between August 2015 and

[1] https://samsung.com/

May 2022. For each bulletin, I collected the list of CVEs, and for each CVE I collected the links of the references, i.e. the web pages with information about commits that fixed the vulnerability. I then scraped all of the references, collecting data about the dates of the commits, and for each CVE I defined the *"security patch complete"* date, i.e. the latest date among all the dates collected from commits referenced by the CVE. This date should indicate when a vulnerability was finally fixed in the source code. Since some CVEs affect closed-source components, not all CVEs that appear in the security bulletin have these references. In total, I collected data on 4349 CVEs, 2062 of which had references to one or more commits.

With this data, I computed the "security patch release delay", i.e. the time it takes from the *"security patch complete"* date to when the vulnerability is published on the bulletin. We can consider this delay to be a time window during which the CVE is known, a fix for the vulnerability is known, but user devices remain vulnerable.

Figure 2.1 shows the "security patch release delay" for CVEs that appeared in Android Security Bulletins between January 2019 and May 2022. Among all these vulnerabilities, I found 117 for which the "security patch release delay" was more than 1 year. Even after removing these outliers, the average delay is more than 84 days. Overall, around 6.88% of CVEs have a publish delay of 6 months or more, which becomes 14.50% if we only consider CVEs for which we have the commit dates.

I also considered that the Android ecosystem is vastly different from the time when the first security bulletin was published, and so this information may not reflect the current situation. Thus, I repeated the analysis taking into consideration only bulletins published after Android 10 was released. I chose this Android version because it is the first release that introduced Project Mainline. With this configuration, the average "security patch release delay" is over 99 days, and the percentage of updates with more than 6 months of "security patch release delay" becomes 9.33%, which grows up to 17.45% if we consider only CVEs that have at least one reference to the AOSP repository. This shows that despite Google's efforts, the situation is still far from ideal, and there is a need for a solution that can be widely adopted by users.

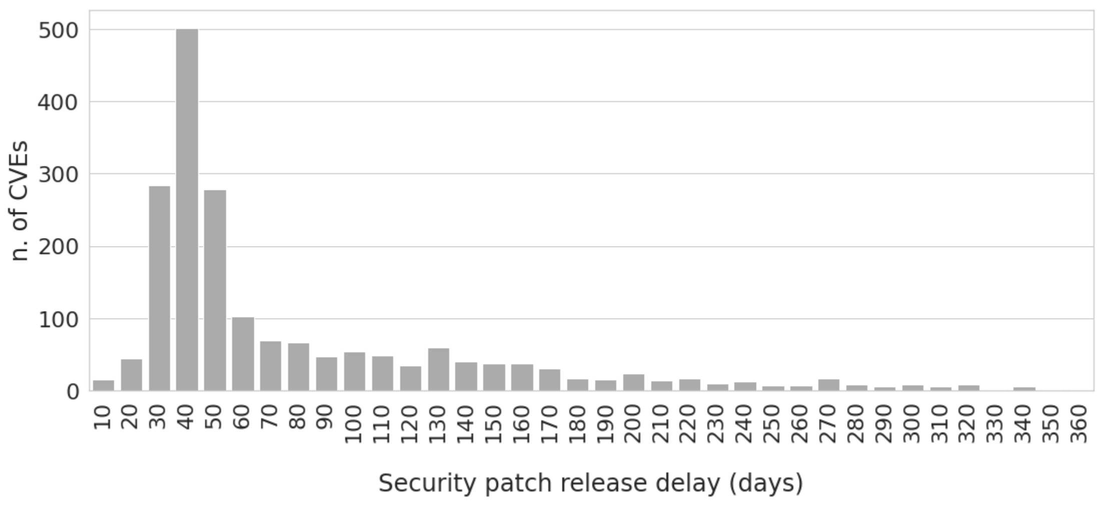


Figure 2.1: Publish delay of CVEs in the Android Security Bulletin.

2.2 Samsung website

To update Samsung devices, users can either connect their device to a computer and use *Smart Switch* [10], or they can install the Over-The-Air (OTA) update directly from their device. On the official website, there is no link to a list of updates, and no way to download the firmware files. However, the protocol used by Samsung devices to communicate with the Samsung Samsung Firmware Update Server (FUS) has been reversed and there are some third-party applications that allow users to download firmwares directly from Samsung servers. For instance, *samloader* [11] is an open-source application written in python that can be used to download the latest firmware for any Samsung Android device, provided that the user knows the model code and the Country Specific Code (CSC) of the device.

I decided to use samloader to collect data about software updates on Samsung devices. Since samloader communicates directly with the FUS, the data collected this way should be trustworthy, i.e. it should reflect what really happens on Samsung devices. However, for each device Samsung only stores the latest update, so samloader cannot be used to get the history of software updates. In addition, while I was only interested in the metadata of the software updates (the date it was published and the security patch level), with samloader I was forced to download the whole firmware archive, which can be up to several gigabytes in size and that I also needed to extract.

While looking at the communication between samloader and the FUS, I noticed that in the response the FUS sends to samloader there is an Uniform Resource Locator (URL) that is unique for each device [2] and points to an *"update notify"* webpage on the official Samsung website, that shows the list of software updates received by the device along with their security patch level and the date in which the updates were published.

I scraped the "update notify" webpage of 386 Samsung devices released between January 2019 and December 2021, collecting data about 6538 software updates that were published between January 2019 and May 2022 [3]. From the data, I saw that on average there are 45 days between subsequent security updates for the same device. I also computed the "patch integration delay", i.e. the number of days from when Google publishes an Android security bulletin to when a security update with the related security patch level is published. In order to take into account the fact that devices that do not have a monthly security update release schedule skip some of the patches, I considered those security patches as being released along with the next security patch. For example, if a device receives an update with security patch level *01-06-2020* on June 2 and then it receives the next update on September 10 with security patch level *01-09-2020*, I consider the updates with security patch level *01-07-2020* and *01-08-2020* to be received on September 10 as well. Also, since there are sometimes multiple updates with the same security patch level, on each device I only considered the first update for each security patch level.

Figure 2.2 shows the "patch integration delay" of the security patches released between January 2019 and May 2022, for Samsung devices released between January 2019 and December 2021: on average, this delay was over 39 days, and 25% of the updates were published with more than 57 days of delay.

2.3 Key takeaway

From the analysis of the data I collected from the Android security bulletin and Samsung, we can see that it takes a considerable amount of time for security patches to reach users, giving malicious actors a large time window they can take

[2]the format of the URL is `https://doc.samsungmobile.com/{model}/{csc}/doc.html`, replacing `model` with the model code of the device and `csc` with the CSC

[3]Appendix A contains the full list of devices used in this study.

advantage of to exploit the vulnerabilities. While in the last years there have been some efforts to improve the situation, there is still much work that needs to be done, and it is important to find solutions that can reduce this time window to a minimum.

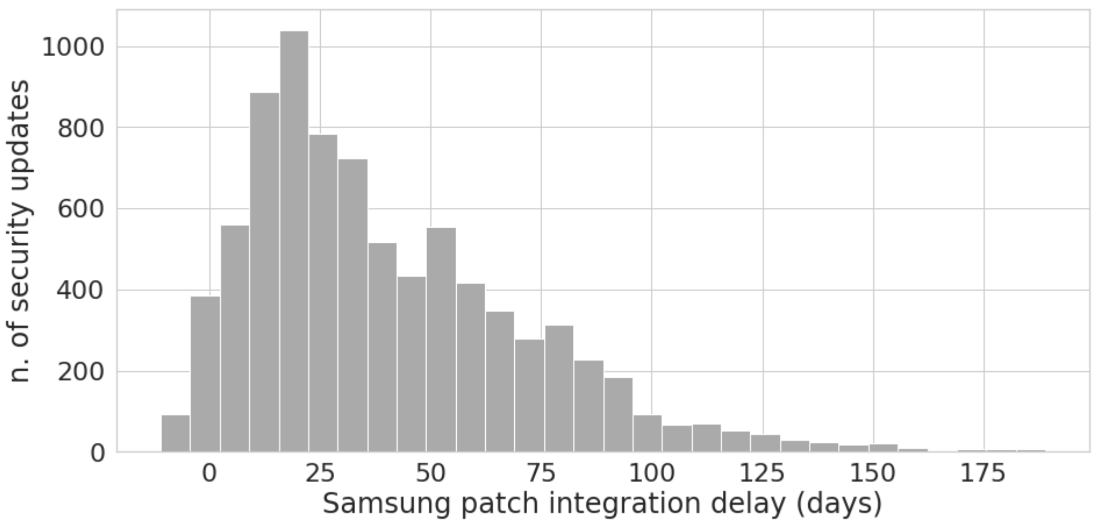


Figure 2.2: Patch delay of software updates.

3 Background

3.1 Layers of the Android architecture

The Android platform is composed of many different components, which can roughly be grouped in layers as shown in Figure 3.1. In the following paragraphs, I describe the layers in terms of which components they contain.

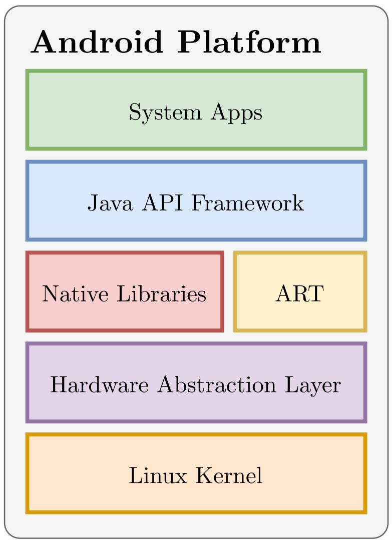


Figure 3.1: Layers of the Android Platform software stack.

System Apps. The *System Apps* layer includes all the basic applications that come with Android, e.g. apps for email, SMS messaging, calendars, and contacts. Although they may have more privileges than third-party applications, such as the privileged permissions [12], users generally can install alternative applications and set them as the default to replace these system apps. A notable exception is the system *Settings* app. Android assigns the User Identifier (UID) 1000 to Settings, and this results in the application having special permissions, such as the ability to send broadcasts to protected `BroadcastReceivers`.

Java API Framework. The *Java API Framework* includes all the Android Java classes that a developer can use when building their applications. These classes provide modular system components and services that are the basic building blocks for most applications and include UI components, proxy classes used to communicate with system services (e.g. the `ActivityManager` class, used to communicate with the `ActivityManagerService`) and *ContentProviders* that application can use to access data managed by system applications (e.g. the contacts saved on the device).

Native C/C++ Libraries. This layer contains Android system core components that contain native code, i.e. libraries written in C or C++ and compiled into binary code. Android provides developers with Java bindings they can use to access some of the functionality of these libraries. An example is the OpenGL ES library, which can be accessed through the Java OpenGL API provided by the Android Framework. Some libraries that are part of this layer are not made available to app developers, but are used internally by the Framework classes (e.g. *libminikin* to compute the app layout).

Android Runtime. In Android, each application runs inside its own process, which contains its own instance of the Android Runtime (ART). ART is the Virtual Machine that executes the Dalvik Executable format and Dalvik Executable format (DEX) bytecode specification and replaces Dalvik in Android 5.0 and later versions. In addition to ART, this layer also includes the core runtime libraries, a set of libraries that provide some of the functionality of the Java programming language. Finally, this layer contains libraries tied to operations such as the system init and reboot.

Hardware Abstraction Layer. This layer provides interfaces for device hardware capabilities that can be used by the Java API Framework layer. Hardware Abstraction Layer (HAL) includes library modules for different types of hardware devices (e.g. camera, Bluetooth), which are loaded by the Android system when the framework API makes a call to access a device hardware. Abstracting the details of hardware access provides a more modular architecture, and makes it easier to implement functionality without affecting the higher layers so that device vendors can update the framework without affecting this layer.

Linux Kernel. Android is an operating system based on Linux and uses a modified version of the Linux Kernel with some additions aimed at mobile devices, such as the *Binder IPC driver* and *Low Memory Killer*, a more aggressive memory management system.

3.2 Virtualization

Virtualization in computing allows the creation of virtual versions of actual resources, such as storage devices, CPUs, or operating systems.

The most traditional type of virtualization is Hardware Virtualization, which abstracts the details of the hardware and allows the creation of virtual machines that behave like a computer with an operating system. This type of virtualization relies on an *hypervisor*, that creates and runs the virtual machines. Virtual machines are isolated and independent from each other, and different virtual machines can have different operating systems installed. Examples of software that uses this type of virtualization are VirtualBox [13] and KVM [14].

In recent years, OS-level virtualization is becoming more popular, in part thanks to the shift to cloud technologies. This type of virtualization is implemented by the operating system, which allows the creation of isolated *containers* and assigns them resources. Compared to virtual machines in Hardware Virtualization, OS-level virtualization happens at a higher level, and all containers share the same underlying operating system, with the advantage of allowing more lightweight virtualization. The most popular example of OS-level virtualization is Docker containers [15].

In Android, neither Hardware Virtualization nor OS-level virtualization is available[1]. However, there are some app-level virtualization frameworks that allow host applications (also called containers) to install and launch guest applications (also called plugin applications or simply plugins) inside a virtual environment they create. App-level virtualization is mainly used by the so-called *dual-instance* applications, which are applications that can be used to run multiple instances of the same application at the same time on a singular device. Users find this type of application particularly useful to log in to multiple social media accounts. The most notable example of an app-level virtualization framework is VirtualApp [19]. In the following section, I describe the main mechanisms VirtualApp uses to create the virtual environment and launch plugin applications.

3.3 VirtualApp

In order to understand how app-level virtualization in Android works, I studied the source code of VirtualApp, which is the most popular open-source virtualization framework, with over 8000 stars on GitHub. To achieve app-level virtualization, VirtualApp employs an architecture with multiple processes that can communicate with each other:

- **Host Application**: this application acts as the container, and handles the installation and launch of plugin applications inside the virtual environment VirtualApp creates.

- **Server Process**: this is a process that contains several "Virtual Services" that VirtualApp creates and that are necessary for the virtual environment to work. An example of such a service is the `VPackageManagerService`, which is used to install applications inside the container and interact with them.

[1]There have been some attempts at implementing OS-level virtualization on Android [16][17], and it looks like Hardware Virtualization support is coming with Android 13 [18], however at the time of writing stock Android devices do not support either form of virtualization out-of-the-box.

- **Guest Application**: VirtualApp launches each guest application inside a dedicated, separate process. Before actually loading the code of the plugin application, VirtualApp sets up the virtual environment inside which it will run.

With VirtualApp, it is possible to install and launch inside a container plugin applications that are not installed on the host operating system. From the point of view of the Android operating system, all plugin applications are part of the same container application, and they all share the same UID. To enable the guest applications to run inside the virtual environment created by the container application, VirtualApp employs a number of different mechanisms.

3.3.1 Dynamic Proxies

Dynamic proxies are provided by the Java language as a part of its reflection functionality. Dynamic proxies are defined through an *invocation handler*, which implements the `invoke` method that is called whenever a method is called on the proxy object. Listing 3.1 shows an example of a simple invocation handler that forwards method calls to the original object. While this example is very simple, it is clear that more complex invocation handlers are very flexible and can change the arguments of the call before forwarding it to the original object or even return some value without calling the method on the original object at all. The `getProxy()` method in the example shows how a dynamic proxy instance can be created from an invocation handler.

Android applications make extensive use of *system services*, to perform operations that go from starting activities to accessing the device sensors. In Android, all communication between applications and services is done through the Binder. To communicate using the Binder Android generally uses proxy classes that are automatically generated from an interface defined using Android Interface Definition Language (AIDL). For instance, the `PackageManager` class, which is defined by the Android Framework APIs, under the hood handles all the communication with the `PackageManagerService` using a proxy class that implements the `IPackageManager` interface.

To load guest applications inside the virtual environment and make it possible for these applications to work and call system services, VirtualApp uses dynamic proxies and injects them into the guest application process, so that they are used instead of the proxy classes when the guest application tries to communicate with a particular system service. This allows VirtualApp to intercept all calls to system services the guest application makes, and if necessary change their arguments. This in turn makes it possible even for applications that are installed only inside the virtual environment, and not in the underlying operating system, to call system services.

```java
import java.lang.reflect.*;

public class MyProxy implements InvocationHandler {

    private Object obj;

    public static Object getProxy(Object obj) {
        return Proxy.newProxyInstance(
            obj.getClass().getClassLoader(),
            obj.getClass().getInterfaces(),
            new MyProxy(obj));
    }

    private MyProxy(Object obj) {
        this.obj = obj;
    }

    public Object invoke(Object proxy, Method m, Object[] args)
        throws Throwable
    {
        return m.invoke(obj, args);
    }
}
```

Listing 3.1: Definition of a simple dynamic proxy.

3.3.2 ClassLoader Hook

A `ClassLoader` in Java is an object responsible for loading classes at runtime. `ClassLoader` itself is an abstract class, and classes that extend this class and implement its methods are platform-dependent. In Android, there are multiple classes that extend `ClassLoader`, but they all inherit from `BaseDexClassLoader`, which defines the common functionality needed to load classes from *dex* and *apk* files. A `BaseDexClassLoader` instance contains a `DexPathList`, which points to a list of `DexFile` objects. On construction, `BaseDexClassLoader` constructs the `DexPathList` from a default library path, i.e. loading DEX files from the application folder. `DexPathList` in turn initializes the `DexFile` objects, which are loaded by calling the native `openDexFileNative()` method. This process is shown in Figure 3.2.

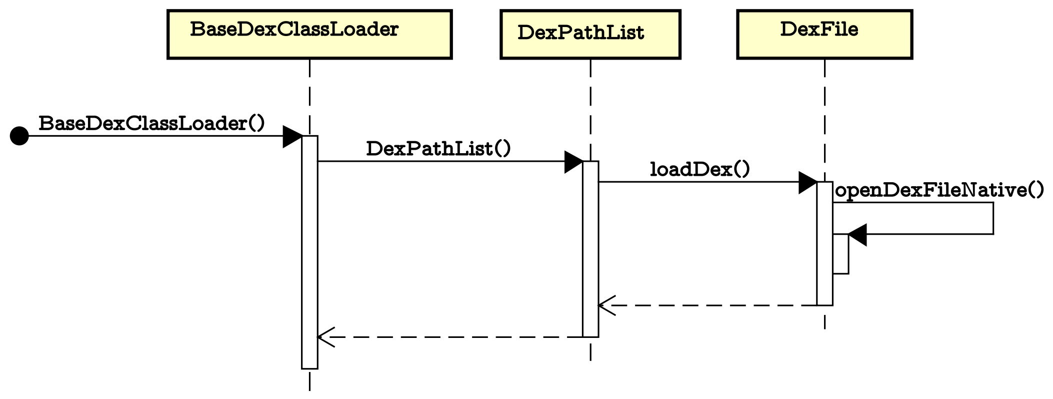


Figure 3.2: Sequence diagram showing how a *dex* file is loaded by *BaseDexClassLoader*.

When a plugin application is installed inside the container, but not on the host operating system, the files that belong to that application are installed in the data folder of the container, and for this reason, the system `ClassLoader` is unable to load the application correctly. VirtualApp solves this issue by hooking the `openDexFileNative()` method so that when it is called it loads the files from the correct folder. To do this, VirtualApp uses Cydia Substrate, a native inline hooking library. Since the original Cydia Substrate requires root, VirtualApp uses a modified version.

3.3.3 Storage redirection

As I mentioned, from the Android operating system point of view a plugin application is just a part of the container application, and all plugin applications share the same UID. This however is in contrast to what happens with usual Android applications, which have unique UIDs that are specific for each application.

Android normally uses the UID to restrict the files and directories that a specific application can access. In particular, each application has its own private data folder, which no other application can access. However, since the container and all the plugin applications share the same UID, they all have access to the same data folder, and this can result in conflicts or errors, since guest applications may try to access their "original" data folder instead of the data folder of the container application.

To address these issues, *VirtualApp* uses native hooks to intercept calls to functions related to input/output (e.g. `open()`, `create()`, etc.) and change their arguments so that they instead target a data folder that they have access to. In particular, *VirtualApp* assigns to each guest application *instance* a private data folder inside the private data folder of the host application so that they do not interfere with each other. This also prevents plugin applications from accessing files of other plugin applications.

Technical details Figure 3.3 gives an overview of how VIrtualApp performs storage redirection: the `startIORelocater()` method calls `redirectFile()` and `redirectDirectory()` repeatedly to initialize the list of storage locations that will be redirected in the guest app, and then it calls `enableIORedirect()` to enable storage redirection. This method in turn causes the native `startUniformer` method to be called, which uses Cydia Substrate to hook all functions related to file input/output (e.g. `openat`, `fstatat`, `link`, etc.). These replaced functions are the ones that perform the actual storage redirection: they use the list of redirected storage locations previously defined to change the arguments they receive before forwarding the requests to the original functions. Figure 3.4 shows an example of how a storage redirection hook works.

3.3.4 Installing a guest application

To be flexible, *VirtualApp* provides two different ways to install a guest application inside a container: cloning it from an application installed in the host operating system, or installing it from an *apk* file the user provides.

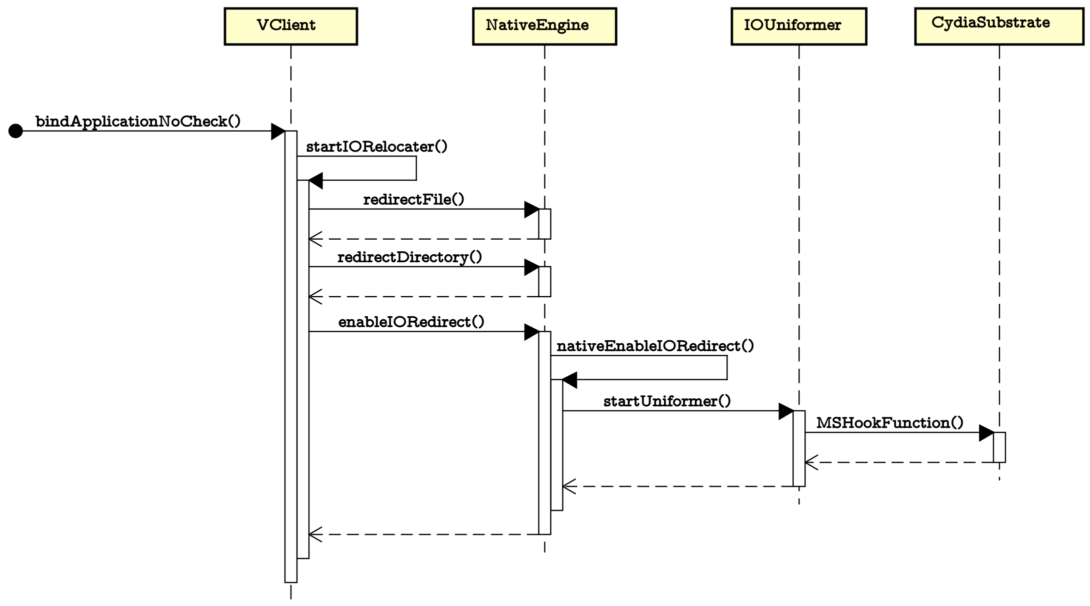


Figure 3.3: Sequence diagram showing how *VirtualApp* redirects the storage of a guest app.

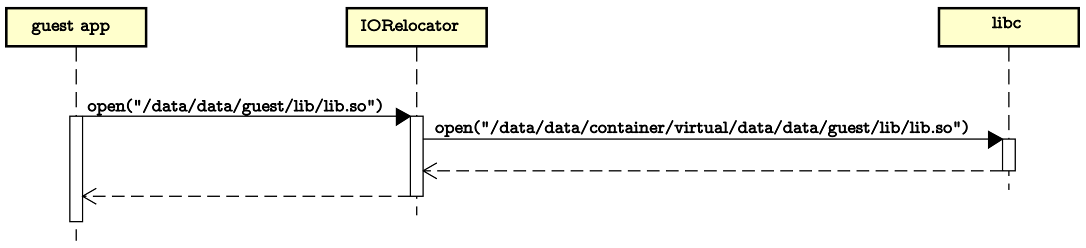

guest app
IORelocator
libc
open("/data/data/guest/lib/lib.so")
open("/data/data/container/virtual/data/data/guest/lib/lib.so")

Figure 3.4: Sequence diagram showing an example of how the storage redirection hook works.

Cloning is particularly useful for *Dual-instance apps*, a kind of application that allows the users to run multiple instances of the same application at the same time. *Dual-instance apps* can be used for example to log in to multiple social media accounts without needing to log out every time to switch accounts. When cloning, the user does not need to find the *apk* file of the desired application, but can just download it from the Google Play Store, without running the risk of downloading a maliciously modified app. Additionally, when cloning an application, *VirtualApp* gives the option to keep the cloned app up-to-date with the updates installed on the host operating system.

However, there are some limitations when cloning. For instance, in older Android versions, some applications could be installed in the host operating system with forward-locking [20]. If this was the case, the *apk* file of the application was stored in an encrypted filesystem image, and it could not be accessed by other applications, including the *VirtualApp* container. While this is no longer the case in recent Android versions, other limitations still apply.

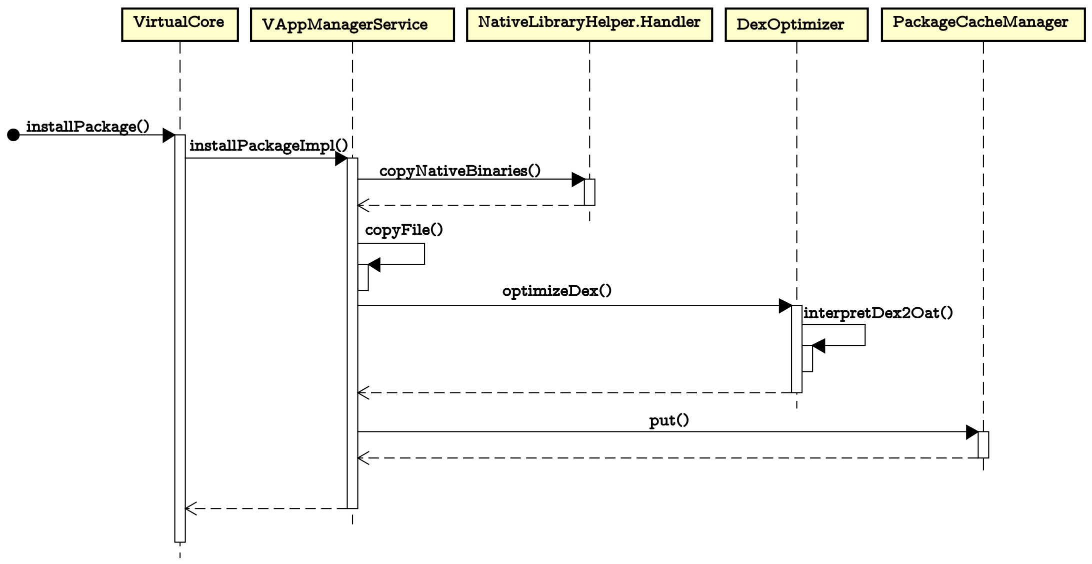
For example, installing an application in the host operating system is required for cloning it, but it may be undesired or even not possible in some situations.

When the user wants to install an application only inside the *VirtualApp* container, they can install it directly from the *apk* file. In such a scenario, *VirtualApp* creates a directory for the package inside the container data directory and copies all the application files inside it. While installing plugin applications from *apk* files is more flexible and can save some space on the device, it opens the door to repackaging attacks: if the *apk* file is obtained from an untrustworthy source, it could be a repackaged version of the desired app containing some malware. In addition, if the user wants to keep the plugin application up to date, they need to manually retrieve and install the *apk* file of the new version of the application every time there is an update.

Technical details Figure 3.5 provides an overview of the steps involved in the installation process. To install a plugin application inside a *VirtualApp* container, the user would typically need to use the `VirtualCore` class and call its `installPackage()` method. The arguments of this method let the user specify a path for the *apk* file of the application and some options which affect how the application is installed. I reported the available options in Listing 3.2.

The `useSourceLocationApk` option is of particular interest. When it is set to `false`, the source *apk* file is copied inside *VirtualApp*'s data directory, so if the original changes or is removed the plugin application is not affected. On the other hand, when this option is set to `true`, *VirtualApp* does not copy the source *apk* file. This is the case when the plugin app is cloned from the host Android operating system and the user wants it to be automatically kept up to date. To retrieve the path to the *apk* file of an installed application the user just needs to use the Android `PackageManager` class.

The `updateStrategy` option dictates how *VirtualApp* behaves when the user tries to install a package that has already been installed inside the virtual environment: it can ignore the new package, compare the versions and install the package only if it is a new version, or overwrite the previously installed package regardless of the version numbers.



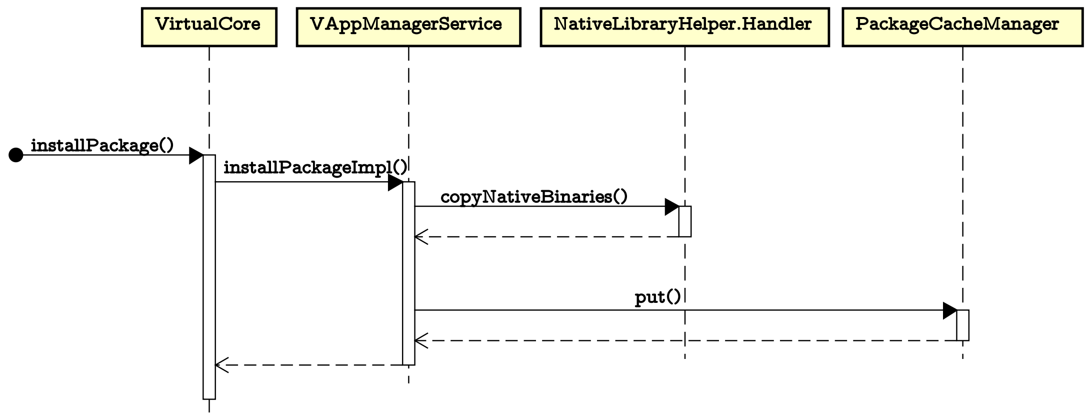


Figure 3.5: Sequence diagram showing how *VirtualApp* installs a guest application from an *apk* file (top) or by cloning (bottom).

The `installPackage()` method forwards the installation request to the `VAppManagerService` in the *VirtualApp* server process, and it is this service that performs the actual installation. The related code can be found in the `installPackageImpl()` method. This method performs the following steps in order:

1. first, it uses Java reflection to access methods of the internal

`NativeLibraryHelper` class, extracting native libraries from the *apk* file and copying them into the dedicated directory.

2. Then, it copies the *apk* file in a dedicated directory inside the *VirtualApp* data directory.

3. After that, the `DexOptimizer` class takes care of invoking `dex2oat` with the correct arguments. `dex2oat` is a command-line tool used by ART to compile *apk* files ahead of time and improve performance and usually is run by the operating system to optimize installed applications.

4. Finally, it updates the `PackageCacheManager`, adding the info and metadata of the application to it, so that the package appears in the list of *VirtualApp* installed packages and can be launched.

```
public class InstallOptions implements Parcelable {
    public boolean useSourceLocationApk = false;
    public boolean notify = true;
    public UpdateStrategy updateStrategy =
        UpdateStrategy.COMPARE_VERSION;

    public enum UpdateStrategy {
        TERMINATE_IF_EXIST,
        FORCE_UPDATE,
        COMPARE_VERSION,
        IGNORE_NEW_VERSION
    }
    // ...
}
```

Listing 3.2: Options that can be passed to *VirtualApp* when installing a package.

If the `useSourceLocationApk` option is true, only the first and the last steps are performed because *VirtualApp* uses the *apk* file in the host operating system directly when launching the plugin application. Using the original *apk file* ensures that when the app is updated in the host operating system the update is reflected inside the container.

To allow users to install multiple instances of the same application, *VirtualApp* provides the `MultiAppHelper` class. Using its `installExistedPackage()` method, the user can create an additional instance of a plugin application that is already installed in *VirtualApp*. Under the hood, *VirtualApp* uses `VUserManagerService` and assigns a different bogus user for each instance. When the (real) user launches a specific instance of the application, the application is launched as if it were launched by the bogus user. Since applications have separated data directories for different users, the instances will be independent of each other. Additionally, this solution has the advantage that all the executable files of a plugin application are shared among instances, saving some space on the device.

After this process is complete, the plugin application is installed inside the container and can be loaded and launched in the virtual environment.

3.3.5 Launching a plugin application

Launching a plugin application inside a container is a complex process. Since the container needs to have total control over the plugin application, using the usual Android APIs to launch it is not possible, since different applications have different UIDs and are generally isolated from each other.

To overcome this limitation, *VirtualApp* declares several *stub Activities*, and then when starting the Activity of a plugin application it finds a free stub Activity and assigns it to that application. The stub Activity then initializes the virtual environment, setting up the storage redirection and the native hooks required to run the plugin application inside the container. This way, the plugin application is run as a part of the container application, with the same UID, and the container has total control over it. Additionally, this allows *VirtualApp* to load guest applications that are not installed on the host operating system.

In addition to stub Activities, the *VirtualApp* container application extends the Android `Application` class, in order to inject *VirtualApp* Dynamic Proxies in the plugin application in place of the original system service classes.

Technical details The steps detailed in this section are summarized in Figure 3.6. To launch an installed plugin app, *VirtualApp* provides the `VActivityManager`. Using the `launchApp()` method the user can specify the package name of the plugin application they wish to launch, as well as the particular instance. This method creates an Intent to launch the requested application and then calls the `queryIntentActivities()` method of `VPackageManagerService` to look for activities that can be launched with that Intent. `VPackageManagerService` resides in the *VirtualApp* server process, and when queried it iterates over the list of activities that the installed plugin applications make available. If any suitable Activity is found, `VActivityManager` sends a request to the `VActivityManagerService` using an Intent to start that Activity. `VActivityManagerService`, which is inside the *VirtualApp* server process as well, finds an unused stub Activity, initializes its process, and wraps the original Intent inside of a new Intent that targets the free Activity. Finally, it calls Android `ActivityManager` to start the stub Activity. The main method related to these steps is `startActivityProcess()` of `ActivityStack`.

All stub activities are instances of the `ShadowActivity` class, which overrides the `onCreate()` method so that it initializes the virtual environment, extracts the wrapped Intent, and starts the Activity of the plugin application. The environment initialization is performed by injecting a custom `Handler` into the current `ActivityThread`. This custom `Handler`, defined in the `HCallbackStub` class, calls `VClient.bindApplicationNoCheck()` on Activity launch. `bindApplicationNoCheck()` does all the initialization work, including hooking native methods and the ClassLoader and redirecting the storage of the plugin application. Then, it creates the actual `Application` using the internal Android class `LoadedApk` through Java reflection. After the custom `Handler` is injected, `onCreate` can just call the `startActivity` method with the extracted Intent, and the new Activity will be started inside a virtual environment.

To inject the Dynamic Proxies that replace Android Services inside the plugin app, *VirtualApp* requires the container to override the `attachBaseContext()` method of `Application` so that it calls `VirtualCore.startup()`. Inside the `startup()` method, the `InvocationStubManager` class is used to first initialize and then inject all Dynamic Proxies inside the plugin application. These proxies will intercept all requests made by the plugin application to Android system services, and change them as needed so that the plugin application works inside the virtual environment.

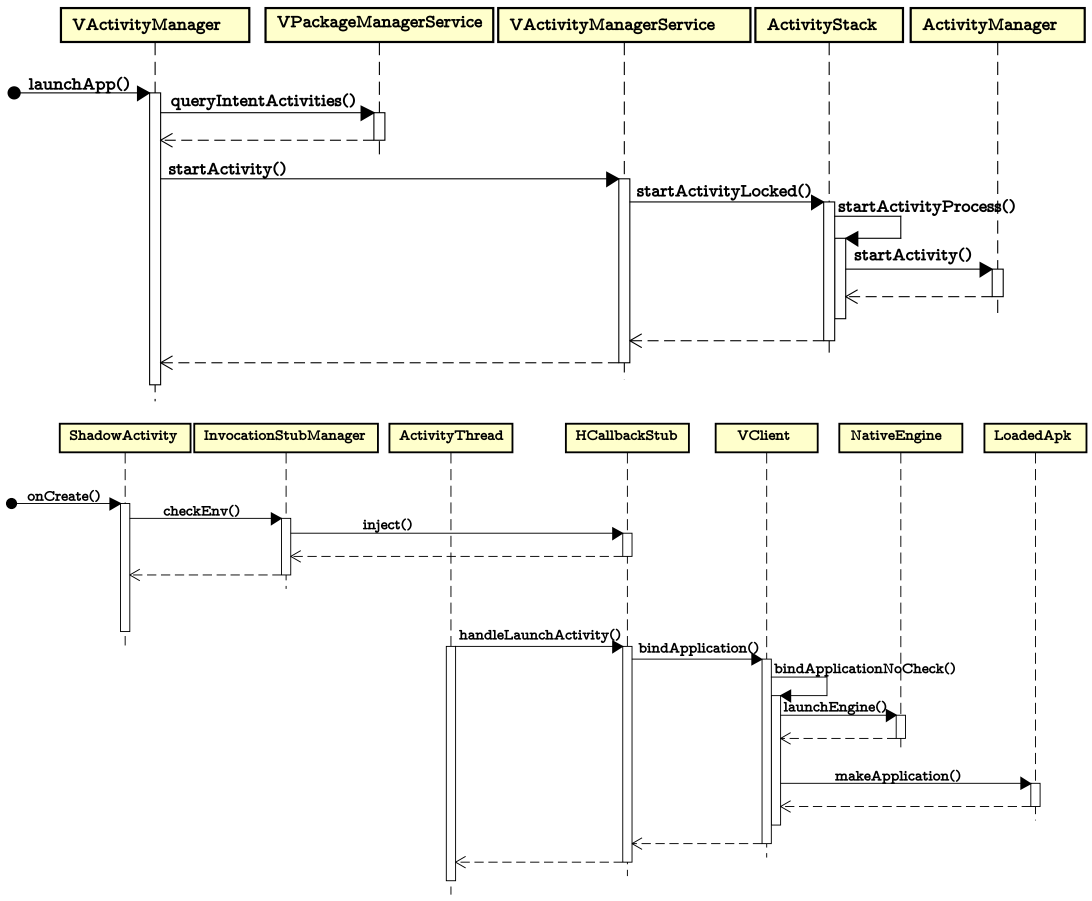


Figure 3.6: Sequence diagrams showing how *VirtualApp* launches a plugin application (top) and creates its virtual environment (bottom).

4
Related Work

In this section, I provide an overview of previous works that are related to my thesis, divided by topic: analysis of Android Security Bulletins (i.e. Section 4.1), solutions to deploy security updates on Android devices (i.e. Section 4.2), and app-level virtualization in Android (i.e. Section 4.3).

4.1 Android Security Bulletins

Android Security Bulletins have been analyzed by previous studies to gain a better understanding of security vulnerabilities in the Android ecosystem. Using the data extracted from the bulletins, researchers analyzed the different vulnerability patterns and the code complexity of the related patches [21], in order to understand system-level vulnerabilities in Android. Other studies focused on the timing aspect of security patches, analyzing the delay in the propagation of kernel security patches from the upstream [22] and the delay with which CVEs appear in vendor-specific security bulletins [23]. In this work, I considered all security vulnerabilities published in Android Security Bulletins, as well as software updates for Samsung devices, to draw the attention to the fact that after a vulnerability is fixed in the Android source code there is a considerable delay before that patch is able to reach user devices.

4.2 Android security patches

It is not uncommon for Android devices to only get sporadic software updates, and in the past, this was even more common. For this reason, many potential alternative ways to deploy security updates to Android devices have been proposed. PatchDroid [24] used a daemon launched at system startup to monitor processes and apply patches to processes using `ptrace`. The PatchDroid application could then be used to install patch libraries on the system. FireDroid [25], which was similar in design, focused instead on enforcing security policies on processes spawned on Android devices by interleaving process system calls and could be used to block OS and application vulnerabilities. More recently, KARMA [26] was proposed as a solution to patch kernel vulnerabilities on Android on-the-fly using a kernel module, adapting the patches to different devices. Similar to KARMA, Embroidery [27] adapts binary code patches to different devices using a system daemon that requires root. InstaGuard [28] patches vulnerabilities using rules that require no code addition, and that can be generated automatically from high-level descriptions of vulnerabilities. Another solution that was proposed to automatically generate patches is VULMET [29], which uses the official patches to generate hot-patches that can be loaded dynamically (e.g. by using PatchDroid). All these solutions have in common the fact that they require either a modification of the Android operating system or root privileges and so they were never extensively employed. Reference Hijacking [30] on the other hand replaces the libraries that an Android process uses by defining a custom *Application* class that customizes Zygote behavior, and so it can be used to patch vulnerabilities in the Framework libraries without modifying the underlying operating system However, it requires the target application to be modified so that it uses the custom *Application* class instead of the default one. Thanks to app-level virtualization, VirtualPatch overcomes these limitations and can be used on stock Android without having to change the code of the guest applications and without root privileges.

Project Mainline, introduced by Google as a part of the Android 10 release, modularizes some of the Android system Components (e.g. Media Codecs and adbd) so that they can be updated from the Google Play Store, without a full system update. In this sense, Project Mainline has goals similar to those of

VirtualPatch. However, only a small part of all the components of the Android architecture are included in Project Mainline. Since the release of Android 10, 1390 CVEs have been published in the Android Security Bulletin, and only 38 of these vulnerabilities have been patched with Google Play system updates (2.73%), so Project Mainline is far from being a solution for the security updates problem. None of the CVEs that I considered in the evaluation of VirtualPatch has been patched via Google Play system updates. This shows that VirtualPatch can be used to patch vulnerabilities that are outside of the scope of Project Mainline.

4.3 App-level virtualization

App-level virtualization in Android was first introduced in 2015 with Boxify [31] and NJAS [32], which independently suggested two different approaches to create a secure sandbox that could be used to run untrusted applications and could provide better permission management than the Android operating system. Afterward, different virtualization engines began to come to light, the most popular of which are *DroidPlugin* [33] and *VirtualApp* [19], and app-level virtualization started to appeal to researchers: some studies described how open-source virtualization frameworks achieved app-level virtualization [34] [35], while others explored the security implications of this technique [34] [36] [37] [38]. None of the previous works have proposed the idea of using app-level virtualization to deploy security patches on Android devices.

5 VirtualPatch

5.1 Design

Android is a complex platform, composed of many different components. Each layer of the Android architecture contains components that have different characteristics, and due to this difference, to patch CVEs that affect different layers we need to use different methods. VirtualPatch is a solution that leverages app-level virtualization to patch vulnerabilities in 4 out of the 6 layers of the Android architecture: System Apps, Java API Framework, Native Libraries, and Android Runtime. The remaining layers (HAL and Linux Kernel) operate at a lower level than app-level virtualization, and so cannot be patched using this technique. In this section I describe the overall design of VirtualPatch (i.e. Section 5.1.1) and the basics of how it can patch vulnerabilities in different layers (i.e. from Section 5.1.2 to Section 5.1.5).

5.1.1 Overview of VirtualPatch

VirtualPatch uses app-level virtualization to dynamically load security patches on Android devices, without requiring a system update or root privileges. Thanks to app-level virtualization, VirtualPatch runs on stock Android OS.

As depicted in Figure 5.1, depending on the layer affected and on the specific CVEs, VirtualPatch can use different techniques to fix security vulnerabilities:

- System Apps patches can either intercept the interaction between the guest app and an Activity of a system app ① or completely replace a vulnerable system app ②;
- Java API Framework patches can either hook Java classes that are part of the Framework ③ or intercept the communication between the guest app and Android System Services ④;
- Native Libraries patches can hook functions and methods of native libraries ⑤;
- Android Runtime patches can hook Java classes that are part of the ART core runtime libraries ⑥.

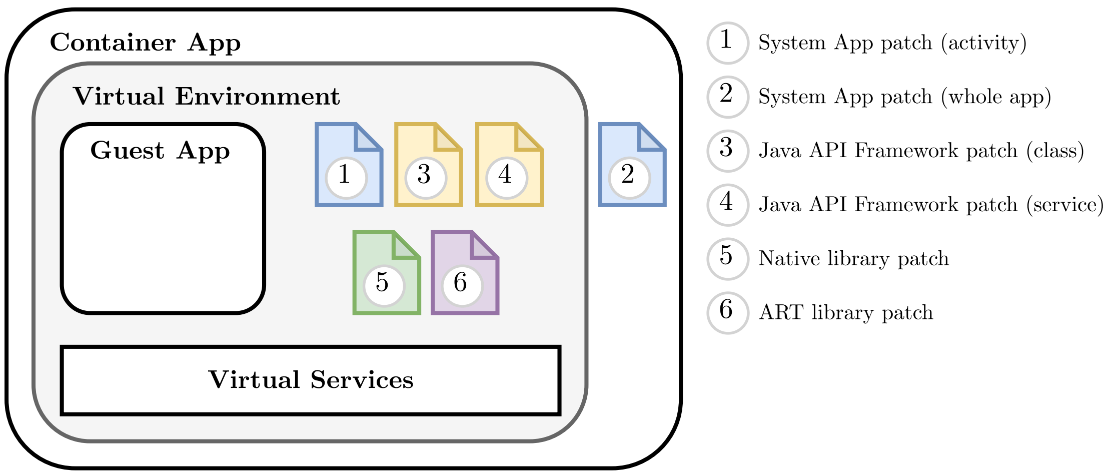

Container App
Virtual Environment
Guest App
1
3
4
2
5
6
Virtual Services
1 System App patch (activity)
2 System App patch (whole app)
3 Java API Framework patch (class)
4 Java API Framework patch (service)
5 Native library patch
6 ART library patch

Figure 5.1: Overview of the architecture of VirtualPatch.

Since the virtual environment creation is a complex procedure that involves multiple steps executed in different processes, different security patches can be loaded at different points during this procedure. Figure 5.2 illustrates the main steps involved in the virtual environment creation, highlighting when the different patches first introduced in Figure 5.1 can be loaded.

5.1.2 System Apps vulnerabilities

CVEs affecting this layer are related to vulnerabilities in the default system applications that are installed along with the operating system, such as Activities that leak data (e.g. *CVE-2021-0444*) or that allow malicious applications to gain additional privileges (e.g. *CVE-2021-0591*, *CVE-2021-0604*). Depending on the specific CVE and on the affected System App, I designed two different approaches to patch vulnerabilities in this layer: **(i)** patching the application and then installing the patched application inside the virtual environment, and **(ii)** checking and sanitizing the Intents used by guest applications to launch system apps.

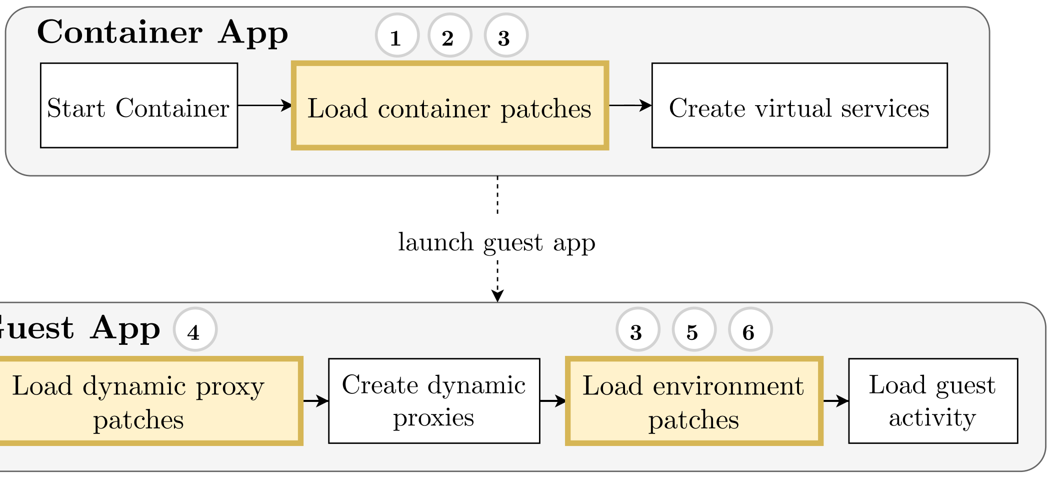


Figure 5.2: Overview of how VirtualPatch creates the virtual environment and loads the different security patches.

The first approach is based on the fact that, when an application tries to start a new Activity, VirtualPatch intercepts the Intent to ensure the new Activity runs inside the virtual environment, and it first looks for a suitable Activity inside the container (i.e. it searches for an app installed inside the container that defines an Activity with an `intent-filter` that matches the received Intent). This results in the patched application being launched instead of the vulnerable application provided by the host operating system.

However, in some situations, it may not be possible to install the patched application inside the container. For instance, since the Settings app has special permissions that cannot be granted to other applications, it is not possible to have an alternative Settings application. The second approach is useful in these situations, allowing VirtualPatch to check and sanitize the Intents used by guest applications before they are forwarded to the operating system. The sanitization could for example block certain Intents or remove some extras from the Intent when VirtualPatch detects that there is an attempt to exploit vulnerabilities in a system application.

5.1.3 Java API Framework vulnerabilities

CVEs affecting this layer can be vulnerabilities in the Java classes provided by the Framework (e.g. *CVE-2019-9376*) or vulnerabilities in a system service (e.g. missing permissions checks), that runs inside a separate process and applications only access through proxy classes (e.g. *CVE-2021-0521*).

For CVEs that affect Java classes that are used in the main application process, a potential patch strategy is to hook the vulnerable methods, replacing them with their fixed versions when the virtual environment is created, before any code of the guest application is executed. Figure 5.3 illustrates how this type of patch works: inside the guest application, all calls to the original (i.e. vulnerable) method are intercepted by the hook installed by the patch, which can for example some validation on the call arguments and decide whether to call the original method or not.

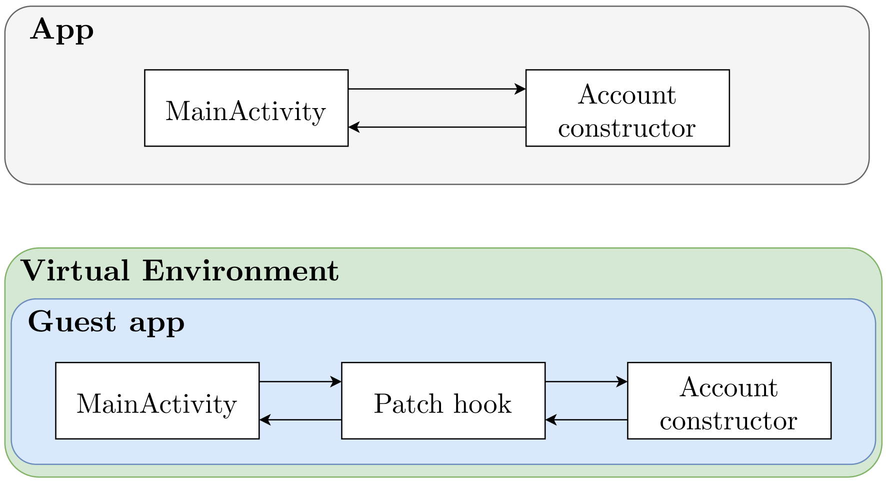


Figure 5.3: Difference between running an app outside (top) and inside (bottom) a virtual environment with a Java hook patch.

To patch CVEs that affect system services, the idea is to intercept the communication between the guest application and the service, checking the requests made to the service and ensuring that no malicious request is forwarded to the actual service. To do this I decided to leverage dynamic proxies. Figure 5.4 illustrates how this type of patch works: while in a normal application the communication with a system service happens directly, in a guest application running inside the virtual environment created by VirtualPatch all the communication with the system service is intercepted by a dynamic proxy. It is the dynamic proxy itself that forwards the requests of the guest application to the system service, potentially changing the arguments or the values returned to the guest app.

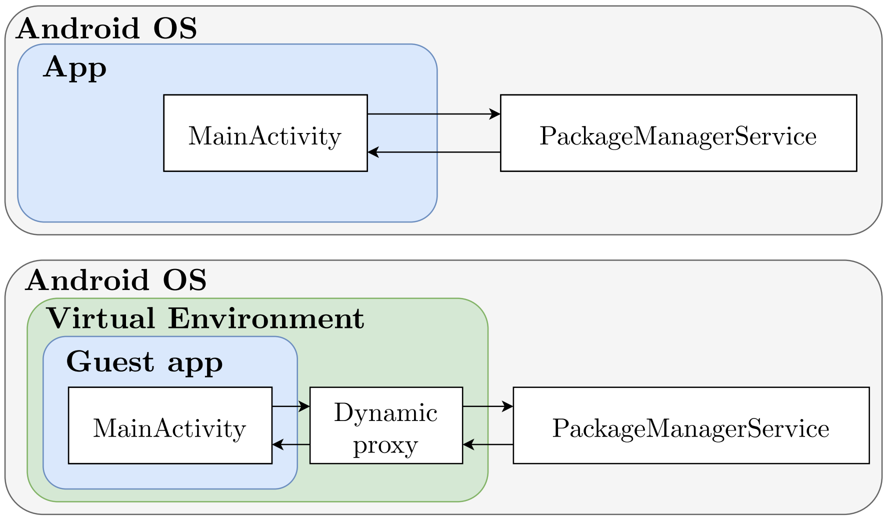


Figure 5.4: Difference between running an app outside (top) and inside (bottom) a virtual environment with a Dynamic Proxy patch.

5.1.4 Native C/C++ Libraries vulnerabilities

CVEs that affect this layer are vulnerabilities in the native code of the libraries that are a part of the layer (e.g. *CVE-2021-0313*). To patch these CVEs, it is possible to hook the affected functions and replace them with patched ones, similar to what I proposed for the *Java API Framework* layer. In this case however both the original and the fixed functions are written in native code, so a different hooking method is necessary, e.g. Procedure Linkage Table (PLT) hooking.

5.1.5 Android Runtime vulnerabilities

CVEs affecting this layer are the least frequent among all CVEs, so it is difficult to get a clear picture of their features. Some affect native code that is executed for example during system reboot (e.g. *CVE-2021-0395*), whereas others affect Java code in the core runtime libraries (e.g. *CVE-2021-0341*). While app-level virtualization naturally cannot be used to patch vulnerabilities that affect the reboot procedure, it can be used to patch CVEs related to Java core libraries. In this case, Java classes and methods can be hooked and replaced, just as described for the *Java API Framework* layer.

5.2 Implementation

In this section I describe the technical details of VirtualPatch, first giving an overview of the overall implementation (i.e. Section 5.2.1), and then describing the techniques that VirtualPatch uses to patch vulnerabilities at the System Apps (i.e. Section 5.2.2), Java API Framework (i.e. Section 5.2.3), Native Libraries (i.e. Section 5.2.4), and Android Runtime (i.e. Section 5.2.5) layers.

5.2.1 Overview

VirtualPatch is based on an open-source version of VirtualApp. I modified how VirtualApp creates the virtual environment to add the functionality needed for all the different types of patches, and I developed a library that offers some APIs that can aid developers in writing security patches for VirtualPatch. Every patch is a standalone file that is loaded at runtime and can either be included as an asset in the apk of VirtualPatch or be loaded from the device's external memory.

For patches written in Java, the library provides the `PatchLoader` class, which defines the lifecycle hooks that other patches can use to specify when the patch is loaded inside the virtual environment, and the `SecurityPatch` class, which provides methods patch developers can use to modify the virtual environment. A patch of this type extends the `PatchLoader` class and is compiled into an *apk* or a *dex* file. Table 5.2 lists the `PatchLoader` hooks that patches can override, and Table 5.1 illustrates the APIs provided by `SecurityPatch`.

For patches written in native code, the patch file is simply a shared library that uses `JNIEnv.RegisterNatives()` to register the `nativeLoad()` JNI method of the `NativeLoader` class. VirtualPatch will call this method to apply the patch.

We can observe that it is possible for multiple patches to target the same method or function. My implementation takes this fact into account and prevents conflicts between patches from happening.

Appendix B shows some code examples that use VirtualPatch APIs to implement security patches.

Method and Description
`void init()`: should be called before any other method to initialize the `SecurityPatch` class.
`void hookJavaMethod(Object target, Method hook, Method backup)`: can be used to hook a java method defined by the `Class target`, possibly saving a reference to the original method in `backup` so that it can be called from `hook`.
`void sanitizeIntent(Method sanitize)`: can be used to add an Intent sanitizer to the virtual environment.
`void addMethodProxy(String className, String methodName, Method proxy)`: can be used to add a dynamic proxy to the virtual environment. `className` indicates the name of the class that defines the method that should be replaced by the proxy, `methodName` indicates the name of the method the proxy should replace, and `proxy` is the new method.
`Object callNextProxy(Object who, Method method, Object... args)`: can be called from a method proxy to call the next proxy in the chain.
`Object[] getProxyArgs(Object[] args)`: can be called from a method proxy to get the arguments used in the original call.

Table 5.1: Summary of `SecurityPatch` APIs.

5.2.2 System Apps

When patching a system app, often it is possible to just install a fixed version of the application inside the virtual environment. Since the source code of system applications is generally available [1], in this case we just need to get the updated code and build the application. This is the most straightforward way to patch some of the CVEs that affect system applications, although it results in a stock version of the application being installed, which may be different from

[1] https://android.googlesource.com/platform/packages/

the one present on the device if the manufacturer customized it. However, most functionality should be the same, so this should not be a problem.

Method	Description
`void onEnvCreate()`	Called in the guest application process during virtual environment creation. Can be used to add Java hooks to the guest application.
`void onDynamicProxyCreate()`	Called before dynamic proxies are created. Should be used to add dynamic method proxies for system services.
`void onServerCreate()`	Called in the server process before all VirtualApp services are created. Can be used to add Java hooks to VirtualApp services and to add Intent sanitizers.

Table 5.2: Summary of `PatchLoader` hooks.

A potential alternative to compiling the app from the source code is to pull the apk of the application from the device, disassemble it, apply the same patch applied in the mainline application (e.g. by modifying the *smali* code), and then recompile it. This procedure needs to be repeated for each different customized version of the application.

The other way CVEs affecting this layer can be patched is by sanitizing the Intents used by apps that run inside the virtual environment, so that potentially harmful Intents are blocked before vulnerable activities are launched and exploited. To implement this kind of patch, I extended VirtualApp's `VActivityManagerService`, adding *Intent sanitizers*.

An Intent sanitizer is a static Java method that receives an Intent to sanitize in input and returns a sanitized Intent. The sanitizer can change the input Intent in any possible way: it can remove some of its *extras*, it can change the action or the target component, and so on. In addition, the sanitizer can also decide to completely block an Intent, preventing the request to start the target Activity from reaching Android *ActivityManager*.

VirtualPatch keeps a list of Intent sanitizers, and individual patches can add their sanitizers to this list. When a guest application tries to start an Activity using an Intent, VirtualPatch "chains" the Intent sanitizers, i.e. it calls the first sanitizer in the list, then uses the Intent returned by this sanitizer to call the second sanitizer in the list, and so on until the last Intent sanitizer in the list is called or one of the sanitizers decides to block the Intent by returning `null`. The Intent obtained at the end of the chain (if any) is the Intent that VirtualApp passes to the *ActivityManager*. With this procedure, if an application tries to exploit a CVE by using a malicious Intent, the malicious Intent will be intercepted by one of the patches and sanitized or blocked.

5.2.3 Java API Framework

As I mentioned, at this layer CVEs affect either Android system services, to which an application can communicate, or Java classes that are used directly by the application.

In the first case, since VirtualApp already needs to intercept and modify some calls to the system services in order for the guest application to work inside the virtual environment, the basic functionality is already provided by VirtualApp itself, in the form of dynamic method proxies. However, VirtualApp dynamic proxies are not designed with security patches in mind, and so they are lacking in some aspects. Most notably, it is not possible to have multiple proxies for the same method, a scenario that is likely to happen when there are many patches. For this reason, I extended VirtualApp dynamic proxies to allow *chaining* multiple proxies. The proxy chain is basically a linked list of method proxies. When the proxy chain intercepts a call to some service, it calls the first method proxy in this linked list. The proxy can perform any operation required by the patch. For instance, it can do additional input validation, and then it can decide to either call the next proxy in the chain, possibly changing the arguments of the call, or interrupt the chain. This process is repeated until either the last proxy in the list is reached, at which point the original method is called, or one of the proxies in the chain interrupts the chain and returns some value without calling the next proxy. This solution is flexible and allows multiple patches to intercept calls to the same method without conflicts.

In the case of CVEs affecting Java classes used directly by the application, the idea is to hook the vulnerable methods and replace them with fixed methods or sanitize their inputs before they are called. For this purpose, I used *YAHFA* [39], a library that can be used to hook Java methods in ART. *YAHFA* modifies ART internal data structures to change pointers to the methods of a class. As illustrated in Figure 5.5, *YAHFA* can be used to replace any method of a Java class with a *hook* method defined in a patch, possibly also saving a reference to the original method in a *backup* method that can be called from the *hook* method. This allows us to write patches that, for instance, only include additional validation, and then call the original method without having to completely reimplement it.

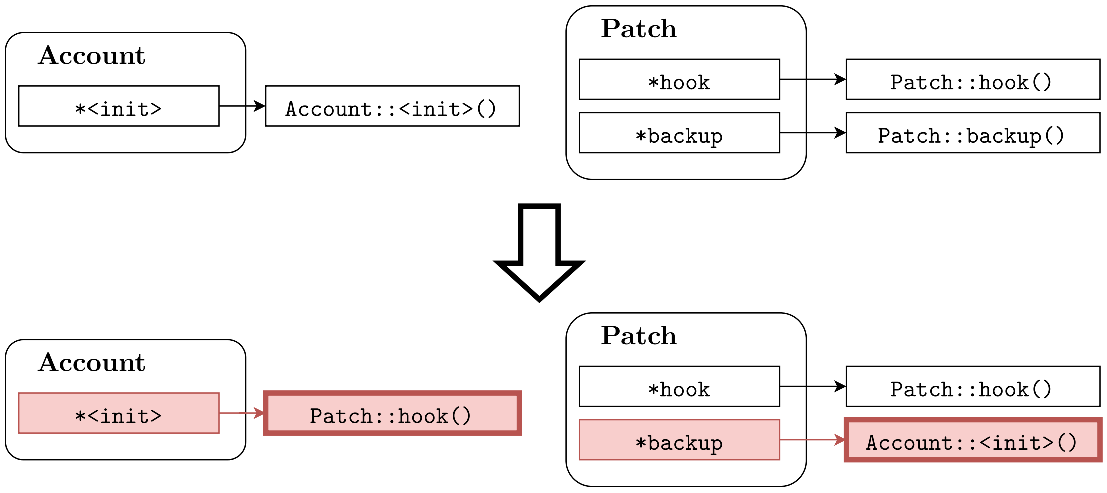


Figure 5.5: Overview of how the Java ART hook works by modifying pointers to Java methods.

5.2.4 Native C/C++ Libraries

Native C/C++ libraries export functions used either directly by an application, or internally by some class of the Framework that in turn is used by an app. When there is a CVE in a native library, virtualization can be used to replace the affected functions in the guest application process, using native hooks. While VirtualApp comes with its own inline hooking framework, a modified version of Cydia substrate [40], it is difficult to use because it is mostly designed for internal use by VirtualApp and is poorly documented. For this reason, I added *ByteHook* [41] to VirtualPatch. *ByteHook* is an Android PLT hook library, which supports all recent Android versions up to Android 12. Additionally, it supports multiple hooks for the same function, so that different patches that target the same function can coexist, and it can hook all the dynamic libraries in the process.

PLT hooking is based on how dynamically loaded libraries are linked by the operating system. in Android, as in Linux, executable files are in Executable and Linkable Format (ELF). An ELF file contains a series of *sections* that can contain different types of data: for example, the code of a program is generally stored in the `text` section, while the information needed for linking is stored in the `dynamic` section. In the context of dynamically linked libraries, there are two ELF sections that are important:

1. `got`: contains the Global Offset Table (GOT), which is a list of entries that point to global variables imported from shared libraries;

2. `plt`: contains the PLT, which is a list of entries that point to functions imported from shared libraries.

To enable lazy loading, i.e. to load the pointers to the shared library functions only if the functions are called, each entry in the PLT points to an entry in the GOT, which at first points to a generic "loading function", and the first time the function is called the GOT entry is overwritten with the pointer to the actual function. Figure 5.6 illustrates what happens when a dynamically loaded function is called from a program. The idea behind PLT hooks is to replace the entry in the GOT so that when the program tries to call that function a different function gets called instead.

Patches for vulnerabilities in the Native Libraries layer can use *ByteHook* to hook native functions. I defined the `NativeLoader` class, a Java class with a *native* method named `nativeLoad()` that native patches are required to implement. To load a native patch, I leverage the fact that libraries loaded with `System.load()` can use `JNIEnv.RegisterNatives()` to override methods defined by libraries loaded previously. Every native patch registers this method, so to apply all patches VirtualPatch can load the shared libraries one at a time, and call `NativeLoader.nativeLoad()` each time.

Using shared libraries directly, we can write the totality of the patch in native code, without the need for verbose Java boilerplate. This approach has also the advantage that the patch files are very small, smaller than the *apk* files that we would need otherwise.

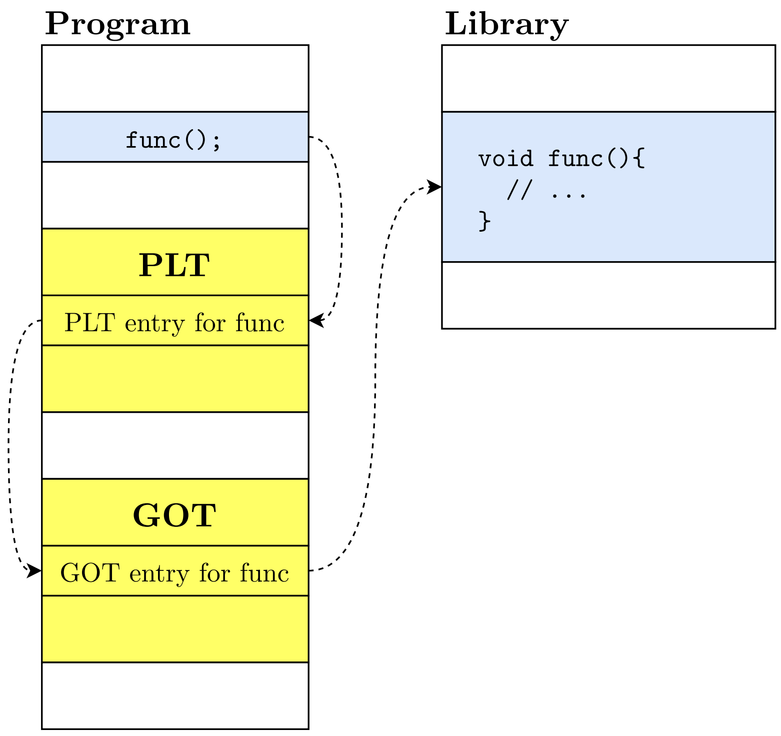


Figure 5.6: Overview of how a function defined in a shared library function is called in ELF executables.

5.2.5 Android Runtime

In the Android Runtime layer, app-level virtualization can mainly be used to patch vulnerabilities in the core libraries. Core libraries consist mostly of Java classes, so patches targeting CVEs in this layer hook Java methods, the same way I described for the *Java API Framework* layer.

6 Evaluation

To evaluate my solution, I picked from past Android Security Bulletins 7 CVEs that affect different layers of the Android architecture, studied the official patches implemented in the AOSP repository, implemented patches for VirtualPatch, and tested them on a real device. To test the patches, I developed proofs-of-concept exploit applications and verified that when the applications are executed inside a patched virtual environment they cannot exploit the CVEs.

The device I used for this evaluation is a Sony Xperia XZ1 smartphone, with Android 9 and 2019-09-01 security patch level. The criteria I used to choose the vulnerabilities are the following:

1. **Security patch level**: since the smartphone I used to test the patches runs an Android build with security patch level 2019-09-01, it is not vulnerable to CVEs that appeared earlier in Android Security Bulletins. Since to test a patch I need to verify that the proof-of-concept application can exploit the vulnerability, I only chose CVEs that appear in Android Security Bulletins since the one published in October 2019.

2. **Android version**: CVEs that are published in the Android Security Bulletin may only affect certain versions of Android. Since the test device has Android 9 installed, I only chose vulnerabilities that affect this version of the operating system.

3. **Layer affected**: to show that VirtualPatch can be used to patch vulnerabilities in different layers of the Android architecture, I chose CVEs that affect all 4 layers (System apps, Java API Framework, Native Libraries, Android Runtime).

6.1 System Apps vulnerabilities

6.1.1 CVE-2021-0604

> "In generateFileInfo of BluetoothOppSendFileInfo.java, there is a possible way to share private files over Bluetooth due to a confused deputy. This could lead to local information disclosure with no additional execution privileges needed. User interaction is needed for exploitation." [42]

Vulnerability Android *Bluetooth* system app exports `BluetoothOppLauncherActivity`, an Activity that third-party applications can use to send files to other devices through Bluetooth, by specifying the *uri* of the file that should be sent. This Activity is part of the *Bluetooth* application, so it can access all the content providers of the same application, including those that are not exported. In particular, it can get files from `MmsFileProvider`, which has access to all files included in MMS messages saved on the device. On vulnerable devices, it is possible to specify an *uri* that points to a file in `MmsFileProvider`, so that a malicious app can bypass permission checks and send that file to another device through Bluetooth.

Exploit I wrote an application that exploits this CVE by using a malicious Intent to start `BluetoothOppLauncherActivity`. The Intent contains an extra called `STREAM` that contains an *uri* pointing to a file in `MmsFileProvider` (e.g. `content://com.android.Bluetooth.map.MmsFileProvider/1`). `BluetoothOppLauncherActivity` shows a list of Bluetooth devices, and when the user selects one device from the list, if the current device is vulnerable, the file will be sent to the other device. Otherwise, an error message will be displayed and no file will be sent.

Security patch The official patch validates *uris* that are used to specify which file to send through Bluetooth and blocks malicious *uris*, i.e. *uris* that identify files of `MmsFileProvider`. To patch this CVE in VirtualPatch, I looked at the code from the official patch and created an Intent sanitizer that worked the same way. The sanitizer checks if the Intent targets the `BluetoothOppLauncherActivity`, and if so it validates the value of the `STREAM` extra, blocking the Intent if a malicious *uri* is detected so that no Activity is started.

6.1.2 CVE-2021-0591

> "In sendReplyIntentToReceiver of BluetoothPermissionActivity.java, there is a possible way to invoke privileged broadcast receivers due to a confused deputy. This could lead to local escalation of privilege with User execution privileges needed. User interaction is needed for exploitation." [43]

Vulnerability The *Settings* app defines `BluetoothPermissionActivity`, which is used to display a confirmation dialog for accepting incoming Bluetooth profile connections from untrusted devices. To send the result of the dialog back (i.e. to tell whoever started the Activity whether the user authorized the connection or not), `BluetoothPermissionActivity` uses a broadcast. The Activity is exported, so it can be called by any application, and only requires the `BLUETOOTH_ADMIN` permission, which Android does not consider "dangerous" thus is requested at install-time. Since the Activity is part of the system *Settings* app, it executes with UID 1000, and is able to access protected *BroadcastReceivers*. On vulnerable devices, it is possible to specify the package and class of the *BroadcastReceiver* that should receive the result, and a malicious app could exploit this to call any protected *BroadcastReceiver*.

Exploit Among protected *BroadcastReceivers*, `MasterClearReceiver` is the one that stands out. This *BroadcastReceivers* is used by the Settings app to initiate a factory reset of the device, normally after the user has confirmed their intentions to do so. I wrote an application that exploits CVE-2021-0591 to call `MasterClearReceier`, with the result that the data on the device gets completely wiped. The application starts `BluetoothPermissionActivity` with a malicious

Intent, with the `PACKAGE_NAME` extra set to `android` and the `CLASS_NAME` extra set to `com.android.server.MasterClearReceiver`. On vulnerable devices, once the user answers to the connection request, `BluetoothPermissionActivity` will call the `MasterClearReceiver` *BroadcastReceiver*.

Security patch The official patch changes `BluetoothPermissionActivity` so that it ignores the two extras of the launch Intent, removing the ability for other applications to specify which *BroadcastReceiver* should receive the result.

To patch this CVE in VirtualPatch I added an Intent sanitizer that checks for Intents which target the `BluetoothOppLauncherActivity`, and removes from them the `PACKAGE_NAME` and `CLASS_NAME` extras. This way, a guest application cannot specify which Activity will receive the result broadcast, and so it cannot make `BluetoothOppLauncherActivity` call `MasterClearReceiver`.

6.1.3 CVE-2021-0444

> "In onActivityResult of QuickContactActivity.java, there is an unnecessary return of an Intent. This could lead to local information disclosure of contact data with no additional execution privileges needed. User interaction is needed for exploitation." [44]

Vulnerability The *Contacts* system app provides `QuickContactActivity`, an Activity which other applications can use to display a dialog that shows information about a given contact. On vulnerable devices, the Activity unnecessarily returns an Intent, possibly leaking contact data.

Exploit The CVE requires specific user interaction for contact data to be leaked. Once the `QuickContactActivity` is launched to display the information about a contact, the user needs to perform the following operations:

1. Open the menu and select the option to link another contact to the current contact;
2. select a contact to link;

3. close the Activity.

Only if exactly these steps are performed, the contact data is leaked. If the user performs any other operation after the contact is linked, there will not be any leak. I wrote an application that detects if any Intent is leaked by the `QuickContactActivity`. If the device is vulnerable and the user interacts as described, the application receives and logs the *uri* of the leaked contact.

Security patch The official patch changes the `onActivityResult()` method of this class, removing the Intent from the returned data.

Exploiting this CVE requires user interaction, and when the Activity is launched there is no way to distinguish between malicious and legitimate Intents, so sanitizing the Intents used to launch this Activity is not possible without preventing all applications from using it. For this reason, I decided to patch the vulnerability by installing the fixed *Contacts* application inside the virtual environment. To patch the application, I pulled the *apk* file from the device, disassembled it using *Apktool* [1], changed the *smali* code so that the *Activity.setResult()* method is called without an Intent, and finally reassembled the *apk* file. *Smali* code can be considered the equivalent of *assembly* for Android DEX files. In this case, I needed to change a single instruction, so it was a viable solution, but patching more complex vulnerabilities with this approach may not be trivial.

6.2 Java API Framework vulnerabilities

6.2.1 CVE-2019-9376

> “In Account of Account.java, there is a possible boot loop due to improper input validation. This could lead to a local denial of service with no additional execution privileges needed. User interaction is not needed for exploitation.” [45]

Vulnerability Android `AccountManager` manages a user’s online accounts. *Authenticators* are services registered with the `AccountManager` that handle the

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

actual storage of the credentials, and other applications can ask permission to access certain credentaisl to the `AccountManager`. For example, an application could ask to get access to the Google or the Facebook account that the user saved on the device. Due to improper input validation, it is possible on vulnerable devices to create an `Account` with an empty name.

Exploit To exploit this CVE, I wrote an application that declares a bogus account type and implements a stub *Authenticator* for this account type. The main Activity of the application then uses Java reflection to create an *Account* with empty name, and calls the `addAccountExplicitly` method of `AccountManager`. On vulnerable devices, the account with the empty name is added by the `AccountManager` to its database of accounts, which becomes corrupted. This in turn causes a boot loop the next time the device is turned on. To declare a new account type and implement its *Authenticator* the application does not need to request any special permission.

Security patch The official patch fixes the `Account` constructor so that it no longer creates accounts with empty names. Android *AccountManager* is a system service. When `addAccountExplicitly` is called by an application, the application communicates with this service through the Binder IPC, and this means that the *Account* to add gets serialized in the application process and deserialized in the service process, i.e. every time the application calls `addAccountExplicitly` the Account constructor is called in the service process. Since the official patch is applied to the whole operating system, it is applied to the *AccountManager* service as well.

The situation is different in VirtualPatch: the container application has total control over the plugin application process, so it can apply the patch to the virtual environment of the guest application, but has no control over the process of the system service. VirtualApp however uses dynamic proxies to replace the `AccountManager` service with its own service (`VAccountManager`, which runs in VirtualApp's server process). For this reason, using Java hooks to replace the `Account` constructor in both the guest application and the server process is enough to patch the vulnerability. Alternatively, the patch could validate the `Account` in the `AccountManagerService` dynamic proxy.

6.2.2 CVE-2021-0521

> "In getAllPackages of PackageManagerService, there is a possible information disclosure due to a missing permission check. This could lead to local information disclosure of cross-user permissions with no additional execution privileges needed. User interaction is not needed for exploitation." [46]

VULNERABILITY `PackageManagerService` is the underlying class used by `PackageManager` to perform most of its operations. `PackageManagerService` itself is not directly available to third-party applications and is only used by classes in the Framework, but it is possible to use Java reflection to access it and call its methods. `PackageManagerService` has a method called `getAllPackages()` that, as the name implies, returns a list of all packages installed on the device. This includes packages installed by other users as well. While accessing this kind of information usually is not possible for applications without root permissions, on vulnerable devices, there is no permission check.

EXPLOIT I wrote an application that uses Java reflection to exploit this CVE. The application uses reflection multiple times to first get the `mPM` field of the `ApplicationPackageManager`, which is the proxy to the `PackageManagerService`, and then to call its `getAllPackages()` method. On vulnerable devices, this call returns the list of all installed applications.

SECURITY PATCH The official patch fixes this vulnerability by allowing only system applications, root, or shell to access this method.

I used a dynamic method proxy to patch this CVE. The proxy does not call the original method, and instead always returns an empty list, so that there is no information disclosure. I verified that the patch works as expected with the exploit application, and confirmed that inside the patched virtual environment the application cannot get the list of all applications installed on the device.

6.3 Native C/C++ Libraries vulnerabilities

6.3.1 CVE-2021-0313

> "In isWordBreakAfter of LayoutUtils.cpp, there is a possible way to slow or crash a TextView due to improper input validation. This could lead to a remote denial of service with no additional execution privileges needed. User interaction is not needed for exploitation." [47]

Vulnerability This CVE affects *libminikin*, a shared library that Android uses to calculate line breaks in *TextViews*. The library is not directly available to third-party applications, but any application that contains a *TextView* uses it. The function responsible for computing word breaks in a string doesn't handle properly Unicode Bidirectional (bidi) control characters, and this can be exploited to cause a denial of service.

Exploit Unicode provides bidi that can be used to change the direction of a part of a string. This is especially useful in the case of strings that include some word or some sentence written in a different language that has a different writing direction (e.g. a string that contains English text that includes a word in Hebrew).

To exploit this CVE, I wrote an application that passes a purposely crafted string `TextView`. The string repeatedly changes the direction of the text using bidi characters, and, on vulnerable devices, this causes the `TextView` to crash the whole application.

Security patch Android official patch adds these bidi characters to the list of characters after which there is a word break, changing the `isWordBreakAfter()` function.

Since `isWordBreakAfter()` is not an exported function of *libminikin*, to patch it I would need to do it with inline hooks. However, this kind of patch would be difficult to port to multiple devices, since the offset of the function could change from device to device, and the function call could even be inlined by the compiler only on some devices, making the issue even more complex. For this reason, I decided to find exported functions that call this function and are used by Android, and hook them instead. I found that there is only one such function, namely `minikin::Layout::doLayoutRunCached()`. I copied this function, replaced the calls to `isWordBreakAfter()` with inline calls to a fixed `isWordBreakAfter()` function, and hooked `doLayoutRunCached()` instead. Since I used PLT hooks instead of inline hooks, which modify the PLT and use the name of the exported function to find the correct offset in the library, this approach should be more portable and work with different devices without major issues.

6.4 Android Runtime vulnerabilities

6.4.1 CVE-2021-0341

> "In verifyHostName of OkHostnameVerifier.java, there is a possible way to accept a certificate for the wrong domain due to improperly used crypto. This could lead to remote information disclosure with no additional execution privileges needed. User interaction is not needed for exploitation." [48]

Vulnerability This CVE affects the *OkHttp* library. Android uses a modified version of *OkHttp* as a part of the core runtime libraries, which implements the functionality for HTTP and HTTPS connections. While applications cannot use the library directly, standard Java classes such as *HttpURLConnection* and *HttpsURLConnection* use it under the hood, since it provides the implementations for these abstract classes. The vulnerability itself consists of some improper input validation in the functions responsible for validating TLS/SSL certificates, which could result in a certificate for the wrong domain being accepted. In particular, the validation converts the hostnames specified in the certificates to lower-case but does not check that the hostname only contains printable ASCII characters. Certain characters when converted to lower-case may result in a name collision, i.e. two different hostnames are treated as if they were the same.

Exploit The Unicode character *"U+212A Kelvin Sign"*, when converted to lower-case, becomes the lower-case letter "k", which is a collision with the upper-case letter "K". If an attacker were able to obtain a certificate from a trusted

Certification Authority (CA) for a domain that uses the *"U+212A Kelvin Sign"* character, that certificate could be used to impersonate a different domain and it would be accepted by vulnerable devices. Due to the need to obtain such a certificate from a trusted CA, I was not able to exploit this CVE, however, I was able to add log messages to the patched methods to verify that they were called instead of the original ones.

SECURITY PATCH Android official patch fixes the input validation, checking that the hostnames on the certificate only contain printable ASCII characters.

To patch this vulnerability, I hooked the affected methods (the two versions of `verifyHostName()`), using reflection to access the internal package (i.e. `com.android.okhttp.internal.tls`). In the hooks I mirrored the verification introduced by the patch, checking that the input hostnames and patterns only contain printable ASCII characters. If that is not the case, the method returns false, and the certificate will be rejected. Otherwise, it calls the original method, matching what happens in the original patch.

6.5 Time Analysis

Previous studies have analyzed the overhead introduced by app-level virtualization itself and concluded that it is considered acceptable by both the users and the Android documentation [49]. Thus, to evaluate the performance of my solution, I focused on measuring whether the loading of the patches in the virtual environment affects the time to launch the guest apps significantly.

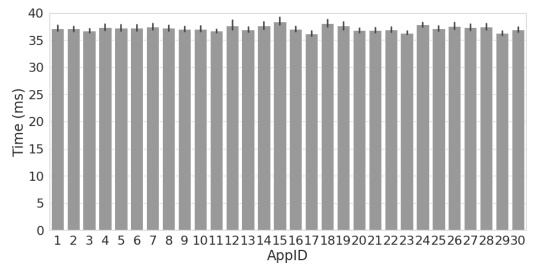
I picked 30 popular applications with more than five million downloads from the *Google Play Store*, installed them inside the virtual environment, and launched them 100 times each, measuring the time it takes to load the patches. The applications are listed in Table 6.1. Since the patches can be loaded at different points in the virtual environment creation and in different processes, as defined by the `PatchLoader` hooks, I used *systrace* [50] to measure the exact time it takes to load the patches defined in the three different lifecycle hooks (`onServerCreate()`, `onDynamicProxyCreate()`, and `onEnvCreate()`). *Systrace* is a tool provided with the Android Platform Tools and can be used to collect and inspect timing information about all processes. I instrumented VirtualPatch code before and after the calls to the functions that load the patches and parsed the traces I obtained to extract only the timings of these sections of code.

AppID	Package	Downloads
1	com.alibaba.intl.android.apps.poseidon	100,000,000+
2	com.contextlogic.wish	500,000,000+
3	com.discord	100,000,000+
4	com.disney.disneyplus	100,000,000+
5	com.duolingo	100,000,000+
6	com.ebay.mobile	100,000,000+
7	com.expedia.bookings	10,000,000+
8	com.facebook.lite	1,000,000,000+
9	com.facebook.orca	5,000,000,000+
10	com.gamma.scan	100,000,000+
11	com.indeed.android.jobsearch	100,000,000+
12	com.instagram.android	1,000,000,000+
13	com.linkedin.android	1,000,000,000+
14	com.mcdonalds.mobileapp	50,000,000+
15	com.meditation.deepsleep.relax	5,000,000+
16	com.netflix.mediaclient	1,000,000,000+
17	com.outfit7.talkingnewsfree	100,000,000+
18	com.pinterest	500,000,000+
19	com.snapchat.android	1,000,000,000+
20	com.tinder	100,000,000+
21	com.ubercab	500,000,000+
22	com.whatsapp	5,000,000,000+
23	com.yahoo.mobile.client.android.finance	10,000,000+
24	com.yahoo.mobile.client.android.mail	100,000,000+
25	com.zzkko	100,000,000+
26	kjv.bible.kingjamesbible	50,000,000+
27	me.lyft.android	10,000,000+
28	org.telegram.messenger.web	1,000,000,000+
29	tv.pluto.android	100,000,000+
30	tv.twitch.android.app	100,000,000+

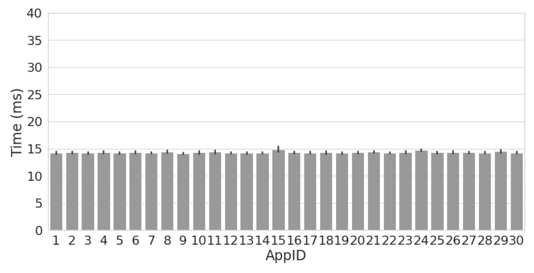
Table 6.1: List of applications I used in the experiments.

Additionally, I used *systrace* to measure the time it takes to “preload” the patches, that is to copy the patch files into the VirtualPatch data folder. While in the prototype VirtualPatch repeats this operation every time it is launched, mainly to collect this data, in a real application this operation would only be performed once during patch installation. Still, even when repeating this operation, it is only executed once when the host application is launched, even if multiple guest applications are launched.

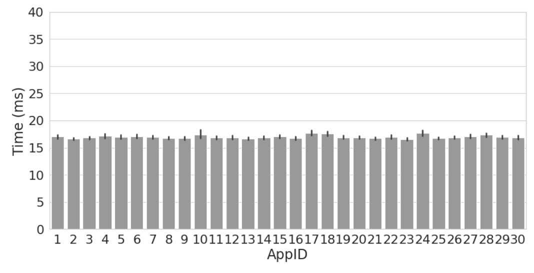
I report the measurements in Figure 6.1. The plot shows the average time VirtualPatch takes to load the patches defined in the three different `BasePatch` hooks and to preload patches, grouping the traces by application. From the plot, we can see that the overhead doesn't change much from application to application. Note that both the *onServerCreate* and the preload hooks are called from the VirtualPatch server process, which is the process in which all virtual services declared by VirtualPatch live, and which is shared across all applications running inside the virtual environment. This means that the overhead introduced by these calls is incurred during the launch of the host application itself, and not during the launch of each guest application.



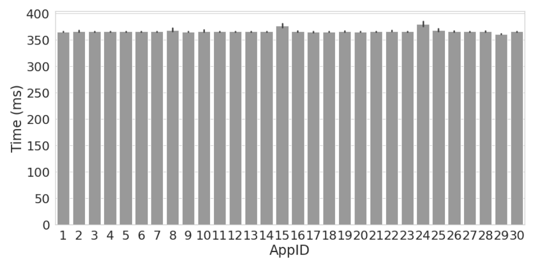
(a) onEnvCreate



(b) onServerCreate



(c) onDynamicProxyCreate



(d) Preload

Figure 6.1: Overhead introduced by patches on different lifecycle hooks, grouped by app.

Table 6.2 summarizes the information about timings across all applications. The "Total (app)" row indicates the total overhead added to guest applications due to patches. As we can see, loading all the patches I implemented takes on average less than 60 milliseconds, which is not enough to be noticed by users.

While for this evaluation I only loaded 7 patches, VirtualPatch is designed as a temporary solution that users can use while they wait for the device manufacturers to publish a system update, so there should not be the need to load thousands of patches at the same time. Even when loading 10 times as many patches as loaded in these experiments, the overhead should remain under 600 milliseconds which I believe to be still acceptable.

Timing	Mean	Std. Deviation
onServerCreate	14.32ms	0.97ms
onEnvCreate	37.18ms	2.20ms
onDynamicProxyCreate	16.98ms	1.44ms
Preload	366.97ms	6.94ms
Total (app)	53.52ms	2.70ms

Table 6.2: Overhead introduced by patches.

7 Conclusion

In this thesis, I analyzed the current Android security patch situation, quantifying the delay that affects the publication of security patches and their integration by Samsung, the biggest Android device vendor. The results of my analysis show that malicious actors have large time windows to exploit known vulnerabilities. In order to address this issue, I studied how Android app-level virtualization works and developed VirtualPatch, a solution that uses this technique to deploy security patches to Android devices without requiring users to do a full system upgrade. VirtualPatch does not require any modification to the Android OS, and can be used on stock Android devices without special permissions (e.g. without root privileges), which are clear benefits over previous solutions. I described the architecture of my solution and detailed the APIs that developers can use to implement security patches that can be loaded by VirtualPatch. Additionally, I evaluated VirtualPatch by implementing security patches for 7 CVEs that were published on past Android Security Bulletins and verifying that these patches were effective in blocking exploit attempts on a vulnerable device. Finally, I evaluated the runtime performance of VirtualPatch and showed that the overhead added by the security patches is negligible from the user's perspective.

While VirtualPatch is a step in the right direction, it still has some limitations in terms of which vulnerabilities can be patched using app-level virtualization. I have shown how this solution is able to patch vulnerabilities in components that are part of the System Applications, Java API Framework, Native Libraries, and Android Runtime layers, however, I believe that VirtualPatch cannot effectively patch vulnerabilities in the Kernel and in the Hardware Abstraction layers.

In addition, since VirtualPatch can only patch applications that users can run inside the virtual environment, there are some vulnerabilities in the layers that I addressed which cannot be patched with VirtualPatch. For example, since the Settings application has special privileges, which no other application can require, we cannot install it inside the virtual environment. Vulnerabilities that can be exploited using the Settings application alone cannot be patched with VirtualPatch.

Moreover, I am aware that current virtualization solutions are not designed with security as the main concern, and that due to this the virtual environment itself may have some vulnerabilities. However, the focus of this work was to show that app-level virtualization can be used to patch security vulnerabilities in Android, and not to investigate the security of current app-level virtualization solutions. In the past, more secure virtualization solutions have been proposed, and while their source code is not freely available, I believe that the techniques I showcased in this work could be easily adapted to those solutions.

A similar argument can be made for the number of applications that VirtualPatch supports: I am aware that not all applications can be executed properly by VirtualPatch, however, this is due to the fact that I based my work on the open-source version of VirtualApp, which has not been updated in months. More up-to-date solutions, including the commercial version of VirtualApp which is being constantly updated, should offer support for most Android applications.

Finally, VirtualPatch is designed to be a temporary solution, i.e. users should use it to install temporary patches while they wait for the device vendor to publish a security update. While loading multiple patches at the same time should not be a problem, as shown by the time analysis, there is a limit to it, and loading too many patches may significantly increase the launch time of applications.

To address these limitations, future research could explore how different types of virtualization (e.g. OS-level virtualization) can be used to patch security vulnerabilities or focus on the creation of a secure app-level virtualization framework that works with recent Android releases and can be used as the basis for further research in this field.

Glossary

AIDL Android Interface Definition Language. 27

AOSP Android Open Source Project. 13, 15, 18, 55

ART Android Runtime. 24, 34, 44, 52

bidi Unicode Bidirectional. 62

CA Certification Authority. 64

CSC Country Specific Code. 19, 20

CVE Common Vulnerabilities and Exposures. 15, 17, 18, 39, 41, 43–52, 54–64, 69

DEX Dalvik Executable format. 24, 29, 59

ELF Executable and Linkable Format. 53

FUS Samsung Firmware Update Server. 19, 20

GOT Global Offset Table. 53

HAL Hardware Abstraction Layer. 25, 43

OS Operating System. 5, 13–15, 25, 26, 40, 43, 69, 70

OTA Over-The-Air. 19

PLT Procedure Linkage Table. 47, 52, 53, 63

UID User Identifier. 24, 27, 30, 35, 57

URL Uniform Resource Locator. 20

References

[1] Ericsson, "Ericsson mobility report november 2021," last access: June 12, 2022. [Online]. Available: https://www.ericsson.com/4ad7e9/assets/local/reports-papers/mobility-report/documents/2021/ericsson-mobility-report-november-2021.pdf

[2] Statcounter, "Mobile operating system market share worldwide," 2022, last access: June 18, 2022. [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide

[3] K. S. Yim, I. Malchev, A. Hsieh, and D. Burke, "Treble: Fast software updates by creating an equilibrium in an active software ecosystem of globally distributed stakeholders," *ACM Trans. Embed. Comput. Syst.*, vol. 18, no. 5s, oct 2019. [Online]. Available: https://doi.org/10.1145/3358237

[4] M. Rahman, "Google's project mainline in android q will help speed up security updates," 2019, last access: June 18, 2022. [Online]. Available: https://www.xda-developers.com/android-q-project-mainline-security/

[5] A. Castro, "Google mandates two years of security updates for popular phones in new android contract," 2018, last access: June 18, 2022. [Online]. Available: https://www.theverge.com/2018/10/24/18019356/android-security-update-mandate-google-contract

[6] Samsung, "Samsung Mobile Security," 2022, last access: June 18, 2022. [Online]. Available: https://security.samsungmobile.com/workScope.smsb

[7] Xiaomi, "Xiaomi Security Updates," 2022, last access: June 18, 2022. [Online]. Available: https://www.mi.com/global/service/support/security-update-1.html

[8] Huawei, “HUAWEI EMUI/Magic UI security updates ,” 2022, last access: June 18, 2022. [Online]. Available: https://consumer.huawei.com/en/support/bulletin/

[9] Google, “Android security bulletin—january 2021 | android open source project,” last access: June 12, 2022. [Online]. Available: https://source.android.com/security/bulletin/2021-01-01

[10] Samsung, “Samsung smart switch,” 2022, last access: June 18, 2022. [Online]. Available: https://www.samsung.com/us/support/owners/app/smart-switch

[11] “samloader,” 2022, last access: June 18, 2022. [Online]. Available: https://github.com/samloader/samloader

[12] “Privileged Permission Allowlisting,” 2022, last access: June 18, 2022. [Online]. Available: https://source.android.com/devices/tech/config/perms-allowlist

[13] Oracle, “Oracle vm virtualbox,” last access: June 18, 2022. [Online]. Available: https://www.virtualbox.org/

[14] “Kernel virtual machine,” last access: June 18, 2022. [Online]. Available: https://www.linux-kvm.org/page/Main_Page

[15] “Docker,” last access: June 18, 2022. [Online]. Available: https://www.docker.com/

[16] W. Song, J. Ming, L. Jiang, Y. Xiang, X. Pan, J. Fu, and G. Peng, “Towards Transparent and Stealthy Android OS Sandboxing via Customizable Container-Based Virtualization,” in *Proceedings of the 28th ACM Conference on Computer and Communications Security (CCS’21)*, 2021.

[17] L. Xu, G. Li, C. Li, W. Sun, W. Chen, and Z. Wang, “Condroid: A container-based virtualization solution adapted for android devices,” in *2015 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*, 2015, pp. 81–88.

[18] M. Rahman, "How google will use virtualization in android 13," 2021, last access: June 18, 2022. [Online]. Available: https://blog.esper.io/android-dessert-bites-5-virtualization-in-android-13-351789/

[19] "Virtualapp framework," 2022, last access: June 18, 2022. [Online]. Available: https://github.com/asLody/VirtualApp

[20] N. Elenkov, *Android Security Internals: An In-Depth Guide to Android's Security Architecture*, 1st ed. USA: No Starch Press, 2014.

[21] D. Wu, D. Gao, E. K. Cheng, Y. Cao, J. Jiang, and R. H. Deng, "Towards understanding android system vulnerabilities: techniques and insights," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, 2019, pp. 295–306.

[22] Z. Zhang, H. Zhang, Z. Qian, and B. Lau, "An investigation of the android kernel patch ecosystem," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 3649–3666.

[23] S. Farhang, M. B. Kirdan, A. Laszka, and J. Grossklags, "An empirical study of android security bulletins in different vendors," in *Proceedings of The Web Conference 2020*, 2020, pp. 3063–3069.

[24] C. Mulliner, J. Oberheide, W. Robertson, and E. Kirda, "Patchdroid: Scalable third-party security patches for android devices," in *Proceedings of the 29th Annual Computer Security Applications Conference*, 2013, pp. 259–268.

[25] G. Russello, A. B. Jimenez, H. Naderi, and W. van der Mark, "Firedroid: Hardening security in almost-stock android," in *Proceedings of the 29th Annual Computer Security Applications Conference*, 2013, pp. 319–328.

[26] Y. Chen, Y. Zhang, Z. Wang, L. Xia, C. Bao, and T. Wei, "Adaptive android kernel live patching," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1253–1270.

[27] X. Zhang, Y. Zhang, J. Li, Y. Hu, H. Li, and D. Gu, "Embroidery: Patching vulnerable binary code of fragmentized android devices," in *2017 IEEE*

International Conference on Software Maintenance and Evolution (ICSME), 2017, pp. 47–57.

[28] Y. Chen, Y. Li, L. Lu, Y.-H. Lin, H. Vijayakumar, Z. Wang, and X. Ou, "Instaguard: Instantly deployable hot-patches for vulnerable system programs on android," in *2018 Network and Distributed System Security Symposium (NDSS'18)*, 2018.

[29] Z. Xu, Y. Zhang, L. Zheng, L. Xia, C. Bao, Z. Wang, and Y. Liu, "Automatic hot patch generation for android kernels," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2397–2414. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/xu

[30] W. You, B. Liang, W. Shi, S. Zhu, P. Wang, S. Xie, and X. Zhang, "Reference hijacking: Patching, protecting and analyzing on unmodified and non-rooted android devices," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 959–970.

[31] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. von Styp-Rekowsky, "Boxify: Full-fledged app sandboxing for stock android," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 691–706. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/backes

[32] A. Bianchi, Y. Fratantonio, C. Kruegel, and G. Vigna, "Njas: Sandboxing unmodified applications in non-rooted devices running stock android," in *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, ser. SPSM '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 27–38. [Online]. Available: https://doi.org/10.1145/2808117.2808122

[33] "DroidPlugin," 2022, last access: June 18, 2022. [Online]. Available: https://github.com/DroidPluginTeam/DroidPlugin

[34] T. Luo, C. Zheng, Z. Xu, and X. Ouyang, "Anti-plugin: Don't let your app play as an android plugin," *Proceedings of Blackhat Asia*, 2017.

[35] L. Zhang, Z. Yang, Y. He, M. Li, S. Yang, M. Yang, Y. Zhang, and Z. Qian, "App in the middle: Demystify application virtualization in android and its security threats," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 3, no. 1, pp. 1–24, 2019.

[36] C. Zheng, T. Luo, Z. Xu, W. Hu, and X. Ouyang, "Android plugin becomes a catastrophe to android ecosystem," in *Proceedings of the First Workshop on Radical and Experiential Security*, 2018, pp. 61–64.

[37] L. Shi, J. Fu, Z. Guo, and J. Ming, "" jekyll and hyde" is risky: Shared-everything threat mitigation in dual-instance apps," in *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, 2019, pp. 222–235.

[38] D. Dai, R. Li, J. Tang, A. Davanian, and H. Yin, "Parallel space traveling: A security analysis of app-level virtualization in android," in *Proceedings of the 25th ACM Symposium on Access Control Models and Technologies*, 2020, pp. 25–32.

[39] "Yahfa," 2022, last access: June 18, 2022. [Online]. Available: https://github.com/PAGalaxyLab/YAHFA

[40] SaurikIT, "Cydia substrate: The powerful code modification platform behind cydia," 2022, last access: June 18, 2022. [Online]. Available: http://www.cydiasubstrate.com/

[41] "ByteHook," 2022, last access: June 18, 2022. [Online]. Available: https://github.com/bytedance/bhook

[42] "CVE-2021-0604." Available from MITRE, CVE-ID CVE-2021-0604, last access: June 12, 2022. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-0604

[43] "CVE-2021-0591." Available from MITRE, CVE-ID CVE-2021-0591, last access: June 12, 2022. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-0591

[44] "CVE-2021-0444." Available from MITRE, CVE-ID CVE-2021-0444, last access: June 12, 2022. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-0444

[45] "CVE-2019-9376." Available from MITRE, CVE-ID CVE-2019-9376, last access: June 12, 2022. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-9376

[46] "CVE-2021-0521." Available from MITRE, CVE-ID CVE-2021-0521, last access: June 12, 2022. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-0521

[47] "CVE-2021-0313." Available from MITRE, CVE-ID CVE-2021-0313, last access: June 12, 2022. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-0313

[48] "CVE-2021-0341." Available from MITRE, CVE-ID CVE-2021-0341, last access: June 12, 2022. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-0341

[49] A. Pham, I. Dacosta, E. Losiouk, J. Stephan, K. Huguenin, and J.-P. Hubaux, "HideMyApp: Hiding the presence of sensitive apps on android," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 711–728. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/pham

[50] "Capture a system trace on the command line | Android Developers," 2022, last access: June 18, 2022. [Online]. Available: https://developer.android.com/topic/performance/tracing/command-line

Acknowledgments

First and foremost, I would like to thank my supervisor, Prof. Eleonora Losiouk, who guided me during my Master's project and assisted me in writing this Master's thesis.

I would also like to thank my family, that has supported me during all these years and made it possible for me to pursue a Master's degree in Computer Science.

A
Samsung Devices

The following table lists the Samsung devices for which I downloaded the software update data as described in Chapter 2.

Model	CSC	Model	CSC	Model	CSC
SM-A426B	EVR	SM-A426B	3IE	SM-A426B	BRI
SM-A426B	XSG	SM-A426B	ROM	SM-E426B	INS
SM-M526B	INS	SM-M526B	CAU	SM-M526B	GTO
SM-M526B	ZTO	SM-M526B	SER	SM-A528B	EVR
SM-A528B	3IE	SM-A528B	BRI	SM-A528B	INS
SM-A528B	GTO	SM-A528B	ZTO	SM-A528B	XSG
SM-A528B	ROM	SM-F926B	EUX	SM-F926B	EVR
SM-F926B	3IE	SM-F926B	CAU	SM-F926B	GTO
SM-F926B	ZTO	SM-F926B	XSG	SM-F926B	SER
SM-F711B	EUX	SM-F711B	EVR	SM-F711B	3IE
SM-F711B	CAU	SM-F711B	GTO	SM-F711B	ZTO
SM-F711B	XSG	SM-F711B	SER	SM-E225F	INS
SM-M325F	INS	SM-A226B	EVR	SM-A226B	3IE
SM-A226B	BRI	SM-A226B	INS	SM-A226B	CAU
SM-A226B	XSG	SM-A226B	SER	SM-A226B	ROM
SM-A226B	TGY	SM-A225F	INS	SM-A225F	CAU

Model	CSC	Model	CSC	Model	CSC
SM-A225F	XSG	SM-A225F	SER	SM-A225F	ROM
SM-A225M	GTO	SM-A225M	ZTO	SM-T730	EUX
SM-T730	KOO	SM-T736B	BRI	SM-T736B	XSG
SM-T736B	ROM	SM-T736B	TGY	SM-T220	EUX
SM-T220	BRI	SM-T220	KOO	SM-T220	INS
SM-T220	CAU	SM-T220	GTO	SM-T220	ZTO
SM-T220	XSG	SM-T220	SER	SM-T220	TGY
SM-T225	EUX	SM-T225	BRI	SM-T225	INS
SM-T225	CAU	SM-T225	GTO	SM-T225	ZTO
SM-T225	SER	SM-T225	ROM	SM-T225	TGY
SM-E5260	CHC	SM-M426B	INS	SM-E025F	INS
SM-F127G	INS	SM-A725F	INS	SM-A725F	XSG
SM-A725F	SER	SM-A725F	ROM	SM-A526B	EVR
SM-A526B	3IE	SM-A526B	ZTO	SM-A526B	XSG
SM-A526B	ROM	SM-A525F	EUX	SM-A525F	INS
SM-A525F	CAU	SM-A525F	XSG	SM-A525F	SER
SM-A325F	INS	SM-A325F	CAU	SM-A325F	XSG
SM-A325F	SER	SM-A325F	ROM	SM-M625F	ZTO
SM-M625F	XSG	SM-E625F	INS	SM-M127G	INS
SM-M022G	INS	SM-A022G	CAU	SM-A022G	SER
SM-G991B	EVR	SM-G991B	3IE	SM-G991B	CAU
SM-G991B	GTO	SM-G991B	ZTO	SM-G991B	XSG
SM-G991B	SER	SM-G991B	ROM	SM-G996B	EVR
SM-G996B	3IE	SM-G996B	CAU	SM-G996B	GTO
SM-G996B	ZTO	SM-G996B	XSG	SM-G996B	SER
SM-G996B	ROM	SM-G998B	EVR	SM-G998B	3IE
SM-G998B	CAU	SM-G998B	GTO	SM-G998B	ZTO
SM-G998B	XSG	SM-G998B	SER	SM-G998B	ROM
SM-A326B	EVR	SM-A326B	3IE	SM-A326B	ZTO
SM-A326B	XSG	SM-A326B	ROM	SM-M025F	INS
SM-A025G	EVR	SM-A025G	3IE	SM-A025G	ROM
SM-A125F	EVR	SM-A125F	3IE	SM-A125F	INS
SM-A125F	CAU	SM-A125F	XSG	SM-A125F	SER

Model	CSC	Model	CSC	Model	CSC
SM-A125F	ROM	SM-A125F	TGY	SM-F415F	INS
SM-F415F	ZTO	SM-G780F	EVR	SM-G780F	3IE
SM-G780F	INS	SM-G780F	CAU	SM-G780F	ZTO
SM-G780F	XSG	SM-G780F	SER	SM-G780F	ROM
SM-G780G	EVR	SM-G780G	3IE	SM-G780G	CAU
SM-G780G	XSG	SM-G780G	SER	SM-G780G	ROM
SM-M515F	INS	SM-M515F	ZTO	SM-M515F	XSG
SM-M515F	SER	SM-M515F	ROM	SM-N980F	EUX
SM-N980F	3IE	SM-N980F	INS	SM-N980F	CAU
SM-N980F	GTO	SM-N980F	XSG	SM-N980F	SER
SM-N980F	ROM	SM-N985F	CAU	SM-N985F	GTO
SM-N985F	XSG	SM-N985F	SER	SM-F707B	ROM
SM-T870	EUX	SM-T870	BRI	SM-T870	KOO
SM-T870	GTO	SM-T870	XSG	SM-T870	SER
SM-T870	TGY	SM-M317F	XSG	SM-M317F	SER
SM-M317F	ROM	SM-M017F	INS	SM-M015G	INS
SM-A217M	ZTO	SM-A716B	BRI	SM-A716B	XSG
SM-A516B	EVR	SM-A516B	3IE	SM-A516B	BRI
SM-A516B	ROM	SM-A215U	SPR	SM-P610	BRI
SM-P610	KOO	SM-P610	GTO	SM-P610	XSG
SM-P610	SER	SM-P610	ROM	SM-P610	TGY
SM-G980F	EVR	SM-G980F	3IE	SM-G980F	CAU
SM-G980F	XSG	SM-G980F	SER	SM-G980F	ROM
SM-G985F	CAU	SM-G985F	XSG	SM-G985F	SER
SM-G985F	ROM	SM-G988B	EVR	SM-G988B	3IE
SM-G988B	CAU	SM-G988B	GTO	SM-G988B	ZTO
SM-G988B	XSG	SM-G988B	SER	SM-G988B	ROM
SM-M115F	BRI	SM-M115F	INS	SM-M115F	XSG
SM-M115F	SER	SM-M115F	ROM	SM-A315F	INS
SM-A315F	CAU	SM-A315F	XSG	SM-A315F	SER
SM-A415F	EVR	SM-A415F	3IE	SM-A415F	CAU
SM-A415F	SER	SM-A415F	ROM	SM-M215F	INS
SM-M215F	XSG	SM-M215F	SER	SM-M215F	ROM

Model	CSC	Model	CSC	Model	CSC
SM-A115F	CAU	SM-A115F	XSG	SM-M315F	INS
SM-M315F	ZTO	SM-M315F	XSG	SM-M315F	SER
SM-F700F	BRI	SM-F700F	GTO	SM-F700F	ZTO
SM-F700F	XSG	SM-F700F	SER	SM-F700F	ROM
SM-F700F	TGY	SM-G715FN	XSG	SM-G715FN	ROM
SM-N770F	EVR	SM-N770F	BRI	SM-N770F	CAU
SM-N770F	GTO	SM-N770F	ZTO	SM-N770F	XSG
SM-N770F	SER	SM-N770F	ROM	SM-N770F	TGY
SM-G770F	EVR	SM-G770F	CAU	SM-G770F	GTO
SM-G770F	ZTO	SM-G770F	XSG	SM-G770F	SER
SM-G770F	ROM	SM-A015F	CAU	SM-A015F	XSG
SM-A015F	SER	SM-A715F	EVR	SM-A715F	3IE
SM-A715F	BRI	SM-A715F	INS	SM-A715F	CAU
SM-A715F	GTO	SM-A715F	ZTO	SM-A715F	SER
SM-A715F	ROM	SM-A715F	TGY	SM-A515F	EVR
SM-A515F	3IE	SM-A515F	BRI	SM-A515F	INS
SM-A515F	CAU	SM-A515F	GTO	SM-A515F	ZTO
SM-A515F	XSG	SM-A515F	SER	SM-A515F	ROM
SM-A515F	TGY	SM-M307F	INS	SM-A207F	INS
SM-A207F	CAU	SM-A207F	XSG	SM-A207F	SER
SM-A207F	ROM	SM-M107F	INS	SM-A707F	INS
SM-A507FN	INS	SM-A307G	GTO	SM-A908B	EVR
SM-F900F	BRI	SM-F900F	GTO	SM-F900F	ZTO
SM-F900F	XSG	SM-F900F	SER	SM-F900F	ROM
SM-A107F	INS	SM-A107F	CAU	SM-A107F	XSG
SM-A107F	SER	SM-A102U	SPR	SM-N975F	EVR
SM-N975F	INS	SM-N975F	CAU	SM-N975F	GTO
SM-N975F	ZTO	SM-N975F	XSG	SM-N975F	SER
SM-N975F	ROM	SM-M405F	INS	SM-G977B	EVR
SM-A6060	CHC	SM-A6060	TGY	SM-A805F	EVR
SM-A805F	3IE	SM-A805F	BRI	SM-A805F	INS
SM-A805F	CAU	SM-A805F	GTO	SM-A805F	ZTO
SM-A805F	XSG	SM-A805F	SER	SM-A805F	ROM

Model	CSC	Model	CSC	Model	CSC
SM-A205F	INS	SM-A205F	XSG	SM-A202F	EVR
SM-A202F	3IE	SM-A202F	ROM	SM-G975N	KOO
SM-G973N	KOO	SM-G970F	EVR	SM-G970F	3IE
SM-G970F	BRI	SM-G970F	INS	SM-G970F	CAU
SM-G970F	GTO	SM-G970F	ZTO	SM-G970F	XSG
SM-G970F	SER	SM-G970F	ROM	SM-A505F	INS
SM-A505F	XSG	SM-A305N	KOO	SM-A105F	INS
SM-A105F	CAU	SM-A105F	XSG	SM-A105F	SER
SM-T720	BRI	SM-T720	KOO	SM-T720	XSG
SM-T720	SER	SM-T510	BRI	SM-T510	KOO
SM-T510	CAU	SM-T510	GTO	SM-T510	ZTO
SM-T510	XSG	SM-T510	SER	SM-T510	ROM
SM-T510	TGY	SM-M305F	INS	SM-M105F	INS

Table A.1: List of Samsung devices used in the study.

B
Patch Examples

This section contains code examples of patches that use different VirtualPatch APIs.

B.1 Intent Sanitizers (CVE-2021-0591)

```java
public class Patch extends SecurityPatch.PatchLoader {
  static final String ACTION =
    "android.bluetooth.device.action.CONNECTION_ACCESS";

  @Override
  public void onServerCreate() throws Throwable {
    SecurityPatch.init();
    Method m = getClass()
      .getDeclaredMethod("sanitizeIntent", Intent.class);
    SecurityPatch.sanitizeIntent(m);
  }

  public static Intent sanitizeIntent(Intent i) {
    Intent ret = new Intent(i);
    String action = ret.getAction();
    if(action != null && action.startsWith(ACTION)) {
      ret.removeExtra("android.bluetooth.device.extra.PACKAGE_NAME");
      ret.removeExtra("android.bluetooth.device.extra.CLASS_NAME");
    }
    return ret;
  }
}
```

B.2 Java Hooks (CVE-2019-9376)

```
public class Patch extends SecurityPatch.PatchLoader {
  @Override
  public void onServerCreate() throws Throwable {
    installPatch();
  }
  @Override
  public void onEnvCreate() throws Throwable {
    installPatch();
  }

  public void installPatch() throws Throwable {
    SecurityPatch.init();
    Constructor<?> target =
      Account.class.getDeclaredConstructor(Parcel.class);
    Method hook = getClass()
      .getDeclaredMethod("hook", Account.class, Parcel.class);
    Method backup = getClass()
      .getDeclaredMethod("backup", Account.class, Parcel.class);
    SecurityPatch.hookJavaMethod(target, hook, backup);
  }

  public static void hook(Account thiz, Parcel in) {
    String name = in.readString();
    String type = in.readString();
    if (TextUtils.isEmpty(name)) {
      throw new android.os.BadParcelableException(
        "the name must not be empty: " + name
      );
    }
    if (TextUtils.isEmpty(type)) {
      throw new android.os.BadParcelableException(
        "the type must not be empty: " + type
      );
    }
    in.setDataPosition(0);
    backup(thiz, in);
  }

  public static void backup(Account thiz, Parcel in) {}
}
```

B.3 Dynamic Method Proxy (CVE-2021-0521)

```
public class Patch extends SecurityPatch.PatchLoader {

  public static List<String>
  getAllPackagesPatch(Object who, Method method, Object... args) {
    return new ArrayList<>();
  }

  @Override
  public void onDynamicProxyCreate() throws Throwable {
    try {
      SecurityPatch.init();
      Method m = getClass()
        .getDeclaredMethod(
          "getAllPackagesPatch",
          Object.class,
          Method.class,
          Object[].class
        );
      SecurityPatch
        .addMethodProxy(PMS, "getAllPackages", m);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

B.4 Native Hook (CVE-2021-0313)

The patched function is the same as the original function, but since `isWordBreakAfter` calls are inlined by the compiler I had to copy it to make it call the patched `isWordBreakAfter`. For this reason the target function is not very interesting, and since it is very long I omitted its code to limit unnecessary clutter.

```
static inline bool isWordBreakAfter(uint16_t c) {
  if (c == ' ' || (0x2000 <= c && c <= 0x200A) || c == 0x3000) {
    return true;
  }
  if ((0x2066 <= c && c <= 0x2069) ||
      (0x202A <= c && c <= 0x202E) ||
      c == 0x200E ||
      c == 0x200F) {
    return true;
  }
  return false;
}

float doLayoutRunCached(/*...*/) {
  // patched function
}

void minikinHook() {
  // full name omitted due formatting constraints
  char* target_name =
    "_ZN7minikin6Layout17doLayoutRunCached...";
  bytehook_hook_partial(
    allow_filter_for_hook_all,
    NULL,
    NULL,
    target_name,
    doLayoutRunCached,
    NULL,
    NULL
  );
}

JNIEXPORT jint JNI_OnLoad(JavaVM *vm, void *reserved) {
  JNIEnv *env;
  jint result = (*vm)->GetEnv(vm, (void **) (&env), JNI_VERSION_1_6);
  if ( result != JNI_OK) {
    return JNI_ERR;
  }
  jclass c = (*env)->FindClass(
    env,
    "dev/sime1/patch/NativeLoader"
  );
  if (c == NULL) return JNI_ERR;
  static const JNINativeMethod methods[] = {
    {
      "nativeLoad",
      "()V",
      (void *) (minikinHook)
    },
  };
  int rc = (*env)->RegisterNatives(
    env,
    c,
    methods,
    sizeof(methods) / sizeof(JNINativeMethod)
  );
  if (rc != JNI_OK) return rc;
  return JNI_VERSION_1_6;
}
```

8 Riassunto in Italiano

8.1 Introduzione

Fin dal suo rilascio iniziale, Android è stato affetto dal problema della *frammentazione*, ossia la presenza di diverse versioni personalizzate del sistema operativo, rilasciate dai diversi produttori di dispositivi. Una delle principali conseguenze di questo problema riguarda la diffusione delle patch di sicurezza, che impiegano molto tempo a raggiungere i dispositivi degli utenti. Ho studiato gli Android Security Bulletin, e analizzato come il più grosso produttore di dispositivi Android (Samsung) gestisce gli aggiornamenti di sicurezza, trovando che dal momento in cui una vulnerabilità viene risolta a quando la soluzione arriva ai dispositivi degli utenti passano spesso diverse settimane. Per questo motivo, ho sviluppato VirtualPatch, una soluzione basata sulla virtualizzazione app-level che può essere utilizzata per proteggere app eseguite su dispositivi Android da vulnerabilità, senza richiedere un aggiornamento dell'intero sistema. Inoltre, per valutare VirtualPatch, ho sviluppato patch di sicurezza ed exploit per 7 diverse CVE, localizzate in diversi livelli dell'architettura di Android, e verificato che tali patch siano efficaci nella difesa contro exploit. Infine, ho misurato il tempo che VirtualPatch impiega a caricare le patch di sicurezza, per sottolineare come queste patch di sicurezza non introducano ritardi significativi nell'esecuzione delle app all'interno di VirtualPatch.

8.2 Motivazione

Google pubblica mensilmente l'Android Security Bulletin, un bolletino nel quale viene fornita una lista di vulnerabilità a cui sono soggetti dispositivi Android e per le quali sono state pubblicate delle patch di sicurezza. Nonostante ogni mese vengano scoperte e pubblicate nel bollettino nuove vulnerabilità, la maggior parte dei dispositivi Android non viene aggiornata con la stessa frequenza. Per comprendere meglio e quantificare il ritardo con cui le patch di sicurezza raggiungono i dispositivi Android, ho studiato il modo in cui Samsung, che è il maggior produttore mondiale di dispositivi Android, gestisce gli update di sicurezza per i propri dispositivi. Ho raccolto dati sugli update di sicurezza Android dai vari Android Security Bulletin, dai repository contenenti il codice sorgente di Android, e dal sito ufficiale Samsung. L'analisi dei dati evidenzia che:

- in media una vulnerabilità appare all'interno di un Android Security Bulletin 84 giorni dopo il relativo commit che risolve la vulnerabilità;
- Samsung impiega in media più di 39 giorni per integrare le patch di sicurezza nella sua versione personalizzata di Android

Questi dati evidenziano come ci sia bisogno di una soluzione che possa accorciare il ritardo tra la scoperta di una vulnerabilità e l'installazione della patch di sicurezza nei dispositivi degli utenti.

8.3 VirtualPatch

Per risolvere il problema sopraccitato, ho progettato VirtualPatch, una soluzione che utilizza la tecnica di virtualizzazione app-level per proteggere dallo sfruttamento di vulnerabilità. VirtualPatch è un'applicazione che gli utenti possono installare nei loro dispositivi Android, e funziona da Container, permettendo agli utenti di installare e lanciare altre applicazioni all'interno di un ambiente virtuale che crea. Inoltre, VirtualPatch permette agli utenti di caricare patch di sicurezza all'interno dell'ambiente virtuale. Le patch di sicurezza sono dei file che vengono caricati dinamicamente da VirtualPatch a tempo di esecuzione. Ho progettato VirtualPatch tenendo a mente l'estensibilità, e definendo delle API che gli sviluppatori possono utilizzare per scrivere patch di sicurezza.

L'architettura di Android contiene molti componenti, i quali possono essere raggruppati in diversi livelli. Componenti appartenenti a diversi livelli hanno caratteristiche diverse, e per questo le patch di sicurezza che agiscono su diversi livelli usano meccanismi diversi.

System Apps. CVE in questo livello sono collegate a vulnerabilità nelle applicazioni di sistema Android. Per sfruttare queste vulnerabilità, è necessario lanciare le relative applicazioni di sistema, quindi ho pensato a due diversi metodi di difesa:

1. installare una versione dell'applicazione non affetta dalla vulnerabilità all'interno dell'ambiente virtuale;

2. sanificare gli Intent usati per lanciare le applicazioni di sistema.

Ho implementato gli “Intent sanitizer”, che possono essere aggiunti dalle patch e che vengono chiamati ogniqualvolta un Intent viene usato per lanciare un'Activity. Gli Intent sanitizer possono controllare gli Intent e rimuovere eventuali dati pericolosi che contengono, oppure bloccare completamente il lancio dell'Activity.

Java API Framework. CVE in questo livello possono essere vulnerabilità nelle classi Java fornite dal Framework, oppure vulnerabilità nei servizi di sistema Android, che sono in un processo separato e a cui l'applicazione accede attraverso classe proxy. Per la scrittura di Patch per le classi del Framework, VirtualPatch utilizza YAHFA, una libreria che modifica le strutture interne di ART per effettuare l'hooking di metodi Java, permettendo in questo modo di sostituire metodi vulnerabili con metodi non vulnerabili. Per la scrittura di Patch per servizi di sistema, VirtualPatch sfrutta i dynamic proxy, uno dei meccanismi chiave della virtualizzazione app-level, estendendoli per evitare conflitti tra diverse patch.

Native Libraries. CVE in questo livello sono vulnerabilità nelle librerie native utilizzate dalle app. In modo simile a quanto descritto per le classi Java, le patch di sicurezza possono fare l'hooking di metodi e funzioni vulnerabili. Siccome in questo caso si tratta di codice nativo, VirtualPatch utilizza un meccanismo diverso per fare l'hooking, ossia PLT hooking.

Android Runtime. CVE in questo livello sono le meno frequenti di tutte, e per questo risulta difficile avere un'idea chiara delle loro caratteristiche. Alcune sono vulnerabilità nel codice nativo di processi come il reboot, con cui patch per l'ambiente virtuale non possono interagire. Altre invece affliggono le "core runtime libraries", librerie Java che implementano certe funzionalità come, ad esempio, le connessioni HTTP e HTTPS. In questo caso, le classi ed i metodi scritti in Java possono essere rimpiazzati usando lo stesso metodo descritto per le classi del Framework.

8.4 Valutazione

Per valutare VirtualPatch, ho scritto patch di sicurezza per 7 CVE che sono apparse in Android Security Bulletin passati. Queste CVE affliggono tutti i livelli dell'architettura Android che ho descritto in precedenza. Per testare le patch, ho sviluppato ed eseguito delle app che implementano exploit per le relative CVE. Ho verificato che all'interno dell'ambiente virtuale creato da VirtualPatch gli exploit falliscono, mentre quando eseguite su un dispositivo vulnerabile (Sony Xperia XZ1 con Android 9 e patch di sicurezza del 2019-09-01) riescono a sfruttare con successo le vulnerabilità.

CVE-2021-0604 L'applicazione di sistema *Bluetooth* esporta l'Activity `BluetoothOppLauncherActivity`, la quale può essere utilizzata da applicazioni di terze parti per mandare dei file ad altri dispositivi utilizzando una connessione Bluetooth. Questa Activity ha accesso a tutti i Content Providers che fanno parte della sua stessa applicazione, e per questo è in grado di utilizzare `MmsFileProvider`, il quale ha accesso a tutti i file ricevuti negli MMS salvati nel dispositivo. Specificando un URI che punta ad un file in `MmmFileProvider`, app malevole sono in grado di accedere a tali file senza i permessi necessari, se eseguite su dispositivi vulnerabili. Ho scritto una patch di sicurezza che usa un Intent sanitizer per bloccare Intent che contengono URI relativi a file di `MmsFileProvider`.

CVE-2021-0591 L'applicazione di sistema *Impostazioni* definisce `BluetoothPermissionActivity`, che viene utilizzata per mostrare un messaggio di conferma per accettare richieste di connessione da parte di dispositivi sconosciuti. Tale Activity utilizza un messaggio Broadcast per notificare della scelta effettuata dall'utente, ed essendo parte dell'app *Impostazioni* ha acceso a Broadcast Receivers protetti. Ho sviluppato un'app che sfrutta questa vulnerabilità per chiamare `MasterClearReceiver`, un Broadcast Receiver che causa la formattazione del dispositivo. La relativa patch utilizza un Intent sanitizer che rimuove alcuni extra pericolosi dagli Intent che hanno come target `BluetoothPermissionActivity`, rendendo impossibile sfruttare la vulnerabilità.

CVE-2021-0444 L'app *Contatti* fornisce l'Activity `QuickContactActivity`, che può essere utilizzata da app di terze parti per mostrare una schermata con informazioni relative ad uno specifico contatto. Nei dispositivi vulnerabili, questa Activity restituisce un Intent non necessario, il quale potrebbe causare la rivelazione di dati sensibili. Per risolvere questa vulnerabilità, ho estratto il file apk dell'app dal dispositivo, l'ho disassemblato, modificato il codice smali facendo in modo che l'Activity non restituisse nessun Intent, ed infine ricompilato ed installato l'app *Contatti* modificata all'interno dell'ambiente virtuale.

CVE-2019-9376 Questa vulnerabilità è causata dalla corruzione di *AccountManager*, resa possibile dal fatto che le app sono in grado di aggiungere utenti con nome vuoto. Questa corruzione fa sì che il dispositivo rimanga bloccato in fase di avvio. La patch che ho scritto effettua l'hooking del costruttore usato per creare l'account, rendendo impossibile che venga passato un'account con nome vuoto a *AccountManager*.

CVE-2021-0521 Il metodo `getAllPackages()` di `PackageManagerService` è un metodo nascosto, il quale restituisce la lista di tutte le applicazioni installate nel dispositivo. Nei dispositivi vulnerabili, tale metodo non effettua nessun controllo sui permessi dell'app che lo chiama, e può essere utilizzato per avere la lista di tutte le app installate da tutti gli utenti presenti nel dispositivo. La patch che ho scritto crea un dynamic proxy che restituisce sempre una lista vuota.

CVE-2021-0313 Questa CVE affligge *libminikin*, una libreria condivisa che Android usa per calcolare il layout delle *TextView*. La libreria non viene usata direttamente dagli sviluppatori di app Android, ma viene usata internamente da alcune classi del Framework. Questa vulnerabilità è causata dal fatto che *libminikin* tratta erroneamente alcuni caratteri Unicode legati a testi bidirezionali, e può essere sfruttata per causare Denial of Service. L'app che ho scritto per testare questa vulnerabilità crea una stringa molto lunga che contiene questi caratteri Unicode e la passa ad una TextView. Nel caso in cui il dispositivo sia vulnerabile, questo causa un crash dell'app. La patch che ho scritto utilizza la tecnica di PLT hooking per rimpiazzare la funzione vulnerabile in *libminikin*.

CVE-2021-0341 Android include la libreria *OkHttp* come parte delle "core runtime libraries", la quale fornisce le funzionalità legate alle connessioni HTTP e HTTPS. Questa vulnerabilità consiste in un errore nella validazione dell'input delle funzioni responsabili della verifica di certificati TLS/SSL, con il risultato che un certificato potrebbe essere accettato per un hostname diverso da quello corretto. La patch che ho scritto effettua l'hooking delle funzioni affette da questa vulnerabilità, aggiungendo i controlli dell'input mancanti prima di chiamare i metodi originali.

Analisi delle prestazioni. Per valutare le prestazioni di VirtualPatch, ed in particolare l'overhead causato delle patch di sicurezza, ho misurato il tempo che VirtualPatch impiega a caricare le patch di sicurezza. Ho installato 30 applicazioni popolari all'interno di VirtualPatch, ed ho lanciato ognuna delle applicazioni 100 volte, utilizzando systrace per effettuare le misurazioni. In media, per caricare tutte le patch VirtualPatch impiega meno di 60ms, ed è quindi impercettibile per gli utenti

8.5 Conclusione

Dati provenienti dagli Android Security Bulletins e dati riguardanti gli aggiornamenti software dei dispositivi Samsung suggeriscono che ci sia una finestra temporale piuttosto larga da quando una vulnerabilità di sicurezza viene scoperta a quando i dispositivi degli utenti sono finalmente protetti. Con VirtualPatch, è possibile installare patch di sicurezza su dispositivi Android velocemente e senza richiedere un aggiornamento completo del sistema, rendendola una soluzione ideale per installare patch di sicurezza temporanee, nell'attesa che i produttori integrino le patch di sicurezza ufficiali pubblicate da Google all'interno dei repository di Android nelle loro versioni personalizzate. VirtualPatch può essere utilizzata per patch di sicurezza a diversi livelli dell'architettura Android, con overhead trascurabile.