**Mascara:**
A Novel Attack Leveraging Android Virtualization

Marco Alecci  
*Department of Information Engineering*  
*University of Padua*  
Padua, Italy  
marcoalecci98@gmail.com

Riccardo Cestaro  
*Department of Mathematics*  
*University of Padua*  
Padua, Italy  
riccardo.cestaro.1@studenti.unipd.it

Mauro Conti  
*Department of Mathematics*  
*University of Padua*  
Padua, Italy  
conti@math.unipd.it

Ketan Kanishka  
*Department of Computer Science and Engineering*  
*Indian Institute of Technology (BHU)*  
Varanasi, India  
ketan.kanishka.cse15@iitbhu.ac.in

Eleonora Losiouk  
*Department of Mathematics*  
*University of Padua*  
Padua, Italy  
eleonora.losiouk@unipd.it

---

**Abstract**—Android virtualization enables an app to create a virtual environment, in which other apps can run. Originally designed to overcome the limitations of mobile apps dimensions, malicious developers soon started exploiting this technique to design novel attacks. As a consequence, researchers proposed new defence mechanisms that enable apps to detect whether they are running in a virtual environment.

In this paper, we propose Mascara, the first attack that exploits the virtualization technique in a new way, achieving the full feasibility against any Android app and proving the ineffectiveness of existing countermeasures. Mascara is executed by a malicious app, that looks like the add-on of the victim app. As for any other add-on, our malicious one can be installed as a standard Android app, that launches our malicious one can be installed as a standard Android app, that looks like the add-on of the victim app. As for any other add-on, our malicious one can be installed as a standard Android app, but, after the installation, it launches a malicious add-on generated by Mascara, the framework we designed and developed to automate the whole process. Concerning Mascara, we evaluated its effectiveness against three popular apps (i.e., Telegram, Amazon Music and Alamo) and its capability to bypass existing mechanisms for virtual environments detection. We analyzed the efficiency of our attack by measuring the overhead introduced at runtime by the virtualization technique and the compilation time required by Mascara to generate 100 malicious add-ons (i.e., less than 10 sec). Finally, we designed a robust approach that detects virtual environments by inspecting the fields values of ArtMethod data structures in the Android Runtime (ART) environment.

**Index Terms**—Mobile Security; Android; Virtualization Technique

---

I. INTRODUCTION

Google Play Store is a broad and wild environment, presently counting more than three millions Android apps [1], published by companies, academies or passionate developers. The openness of the Android platform encourages many people to contribute to the growth of the Play Store. Unfortunately, some of those people exploit it as an attack vector to disseminate malwares. Google has adopted different defence mechanisms (e.g., Google Bouncer [2] introduced in 2012; Google Play Protect [3] released in 2017) to verify the apps immediately after their publication or update. Despite those efforts, malware developers still manage to overcome such detection mechanisms, as it is confirmed by the 870,617 malicious installation packages found in 2019 according to this Kaspersky Lab analysis [4].

Among the different approaches used to design a malware, attackers have recently started exploiting the virtualization technique. When used in the Android Operating System (OS), this technique gives the opportunity to an app (i.e., Container App) to define a new environment, separate from the Android default one, that is able to run any app (i.e., Plugin App), while fully preserving its functionalities. The virtualization technique relies on the dynamic code loading and on the dynamic hooking. The dynamic code loading enables an Android app to bypass the 64K reference limit [5] by loading the external code contained in a Dalvik EXecutable (DEX) file, a Java ARchive (JAR) file or even in the Android Package (APK) of another app (the Android app bytecode can reference up to 65,536 methods, among which the Android framework methods, the library methods and the app own methods). The dynamic hooking allows an Android app to intercept at runtime any call to the Android framework. This approach is not officially supported by Google, but several solutions are already available [6], [7], [8].

The popularity of the virtualization technique, confirmed by the number of downloads of the apps on the Play Store implementing it [9], [10], [11], is given by its main use case scenario, i.e., running multiple instances of the same app on a single device: if a user has two separate Telegram accounts and wants to use them simultaneously, he can have the first running in the Android environment and the second in one of the virtualized contexts. The same goal might be achieved by creating different users on the same device through the Multi-user system behavior, introduced in Android 5.0 [12]. However, this approach has a significant usability limitation: while using a specific user account, the notifications of apps running under a different user are not displayed.
Previous works [13], [14], [15], [16], [17] have already identified attacks exploiting the virtualization technique and even very popular apps have already been the target of new virtualization-based malwares [18], [19]. Thus, researchers [20], [17], [16], [15], [21] have soon started designing novel detection mechanisms, specifically customized to detect virtual environments, which are different from the more well-known emulated environments. However, the attacks identified so far rely only on built-in features of the virtualization technique, provided by all frameworks supporting the generation of Android virtual environments [22], [23], [24]. On the contrary, Mascara is the first attack that relies on a customization process applied to the original virtualization framework and repeated everytime for each single victim app. Such customization process not only enhances the probability to lure users to install the app that will launch Mascara, but it also allows to bypass the existing detection mechanisms for virtual environments. Moreover, Mascara is the first attack that can be launched against any Android app and the framework we developed, Mascarer, automatically generates the malicious add-on of a victim app. Thus, Mascara highlights new features of the virtualization technique that can be exploited to generate undiscovered attacks, even more threatening than the existing ones. Mascara is launched by a malicious app, that is customized to look like the legitimate add-on of a popular app, while it is responsible for the generation of a virtual environment. Its purpose is to extract sensitive data from the victim smartphone by leveraging on the generation of a virtual environment, where the victim app and a malicious app run simultaneously.

Contribution.
- We designed and developed Mascara, a novel virtualization-based attack that bypasses existing defence mechanisms, and Mascarer, a framework for the automatic generation of the malicious add-on launching Mascara;
- We evaluated the effectiveness of Mascara against Alamo, Amazon Music and Telegram and the runtime overhead introduced by the virtual environment. We also evaluated the efficiency of Mascarer by measuring the compilation time required to generate the malicious add-ons of 100 popular apps;
- We executed the existing defence mechanisms (i.e., AntiPlugin [21] and DiPrint [17]) in Mascara and proved that our attack is resilient to such approaches;
- We finally developed a robust detection mechanism to detect Android virtual environments by inspecting the fields values of ArtMethod data structures in the Android Runtime (ART) environment.

Organization. The paper is organized as follows: in Section II, we describe the virtualization technique applied to the Android OS, as well as the features of the DroidPlugin framework used to design Mascara; in Section III, we introduce the related work, while in Section IV, we illustrate the Mascara attack. In Section V and Section VI, we describe the design and implementation of the Mascarer framework; in Section VII, we evaluate the efficiency of Mascarer and the effectiveness of Mascara. In Section VIII, we prove that Mascara is resilient to the existing defence mechanisms. Finally, in Section IX, we discuss about limitations and illustrate our new detection mechanism, before concluding the paper.

II. BACKGROUND ON ANDROID VIRTUALIZATION

Android is a multi-process, Linux-based OS, that executes apps in isolated processes, giving each one a unique Linux UID, its own address space and a private directory. By enabling apps to expose a virtual environment in which other apps can run, the virtualization technique [21], [14] breaks the current Android security model. DroidPlugin [22], VirtualApp [23] and DynamicAPK [24] are the most well-known frameworks supporting the virtualization technique and they differ according to whether their virtual environment can:
(i) run one or multiple apps simultaneously; (ii) run apps not installed on the device; (iii) run any Android app or only a specific set of apps.

To design and develop Mascara, we evaluated all above-mentioned frameworks and we chose to use DroidPlugin, since it allows to build the most damaging attack in comparison to the other frameworks: its virtual environment can execute any Android app, even multiple ones at the same time, including those not installed on the device, and it can be installed as a generic Android app, without requiring additional privileges enabled on the device (e.g., root privileges). In Fig. 1, we show a smartphone with two Android apps using DroidPlugin to generate each one its own virtual environment, in which other apps can run.

![Fig. 1. Multiple virtual environments created through DroidPlugin.](image)

Fig. 2 shows the internal architecture of an app generating a virtual environment through the DroidPlugin framework. The architecture involves the following components: a Container App, several Plugin Apps and a Dynamic Proxy API Module. The Container App is the main component, since it is responsible for the virtual environment generation through the dynamic code loading and the dynamic hooking. The Plugin Apps are the apps running in the virtual environment, while the
**Dynamic Proxy API Module** contains the logic to implement the dynamic hooking through the Java Dynamic Proxy [25].

![Diagram](image)

**Fig. 2. DroidPlugin internal architecture.**

Once the **Container App** has created the virtual environment, it forks its process (i.e., \( \text{pid0} \)) into as many processes as the **Plugin Apps** that it is going to run. The **Container App** and the **Plugin Apps** will share the same UID (i.e., \( \text{uid0} \)), but each one will run in its own process. Then, the **Container App** loads the executable files of the **Plugin Apps**, saved in their APK files, and execute them in their dedicated processes. During the runtime execution, the **Container App** has to manage the lifecycle of **Plugin Apps** components. Android apps contain several components (i.e., **Activities**, **Services**, **Broadcast Receivers** and **Content Providers**), which need to be declared in the `AndroidManifest.xml` file so that the Android OS can register them at the installation time. The management of each component involves an interaction between the Android app and the Android OS: for example, to show an **Activity** on the screen of the smartphone, an app has to send a request to open that **Activity** to the Android OS and it has to receive a reply containing the details of the device where the app is running (e.g., the dimension of the screen). When the **Plugin Apps** run in a virtual environment, the Android OS receives all requests sent by them as if they were sent by the **Container App**. Thus, the **Container App** is expected to declare the same components as the **Plugin Apps** running in its virtual environment. Thanks to the **Dynamic Proxy API Module**, DroidPlugin is able to execute any Android app without customizing each time the **Container App**. As a matter of fact, the **Container App** first declares a generic number of stub components and, then, it can rely on the **Dynamic Proxy API Module** to intercept each request/reply going towards or coming from the Android OS to dynamically change the name of the component. This approach is used for **Activities**, **Services** and **Content Providers**, while **Broadcast Receivers** are parsed from the **Plugin App** `AndroidManifest.xml` file and dynamically registered at runtime. The same approach is also followed for the permissions: the **Container App** declares all existing Android permissions.

**III. RELATED WORK**

Previous works address the Android virtualization technique from three different points and can be divided in three groups, accordingly: the ones focusing on the exploitation of the virtualization technique to develop new attacks; the ones proposing novel detection mechanisms for virtual environments; the ones proposing the virtualization technique as a novel defence mechanism.

**Attacks exploiting Android virtualization.** In a virtual environment, the attacks that can be performed are usually executed by the **Container App** or by the **Plugin App** [13], [15], [17], [16]. Concerning the former, previous works identified the following exploitations:

- The **Container App** creates a virtual environment, where both benign and malicious apps are executed;
- The execution of an app running in the native Android OS is stopped and resumed within the virtual environment under the control of the **Container App**;
- The **Container App** might download malicious code and dynamically execute it at runtime;
- The **Container App** applies new hooks at runtime on the **Plugin App** to make it execute malicious code (i.e., the new repacking attack).

Concerning the **Plugin App**, the possible attacks are as follows:

- **Plugin Apps** can access files of each other;
- A **Plugin App** shares the UID with the **Container App**, which means also the same permissions;
- A malicious **Plugin App** can tamper with the executable files or encrypt files of another **Plugin App**;
- A **Plugin App** can launch a phishing attack to steal user input provided to another **Plugin App** running in foreground.

In addition to the above-mentioned attacks, two popular apps were already found to be the target of a virtualization-based attack. In the first one [18], a dual-instance app enables users to run a second instance of Twitter. Besides creating a virtual environment through **VirtualApp**, the malware hijacks user inputs. The second malware aims at distributing a malicious, updated version of the WhatsApp app [19]. This is definitely similar to the original one, besides the additional malicious code crafted to steal users’ sensitive data. After bypassing the Play Store checks, the developer managed to release his fake version with the same publisher name as the original WhatsApp (i.e., WhatsApp Inc.) and to have it downloaded more than one million times before its removal from the Store.

All the attacks previously illustrated exploit the built-in features provided by the existing frameworks that support the generation of Android virtual environments. Thus, such attacks are not robust enough and can be detecting by anti-virtualization approaches.

**Android virtualization detection mechanisms.** The defence mechanisms proposed so far [21], [17], [16], [15], [20] detect virtual environments by inspecting the following features: the permissions declared by the **Container App** and shared with **Plugin Apps**; the number and names of the components declared by the **Container App**; the app process...
name; the information about the internal storage; the data sharing among Plugin Apps and the stack trace of the Plugin App. All such approaches are not effective, since a malicious Container App can easily introduce new hooks on the Android and Java API, as well as on the native code, to modify their return values.

Other usages of Android virtualization. NJAS [26] and Boxify [27] provide two interesting defense mechanisms against Android malware. The first one builds a virtual environment where malwares run, thus preventing them from gaining control of the user phone. Boxify [27] builds sandboxed environments, where malicious apps run in isolated processes, thus, preventing apps from requiring permissions and accessing the associated resources. Other previous works used virtualization to monitor the apps runtime behaviour [28], [29], [30], to enhance the security level of virtual machine used virtualization to monitor the apps runtime behaviour [28], and accessing the associated resources. Other previous works used virtualization to monitor the apps runtime behaviour [28], [29], [30], to enhance the security level of virtual machine [31] or to support the hot patch feature [32].

IV. Mascara ATTACK

In this section, we provide an overview of Mascara (Section IV-A) and a detailed description of the internal architecture of the malicious add-on (Section IV-B).

A. Overview of Mascara

Mascara refers to the Venetian-Italian word, which means mask and recalls the main purpose of our malware: stealing sensitive data from a smartphone by luring the user to install a malicious app, which looks like a legitimate one. Mascara is launched by a malicious app, which is customized to look like the add-on of a popular app. Our choice to use the add-on technology as a vector for Mascara is motivated by its numerous advantages. Generally speaking, add-ons are accessory components aimed at providing new functionality to existing systems. They have their own, separated logic, but they also strongly depend on the existing systems, without which they can not even work. In the Android environment, the add-on technology follows the same approach: there is a base app, which is itself a standalone app, and a variable number of add-ons, each one providing new features to the base app. Add-ons are Android apps, that have to run in the same mobile device, where the base app is installed, and that can be developed by different Android programmers with respect to the base app ones. An illustrative example of the add-on technology is provided by the “Locus Map Free” app [33], available on the Google Play Store. The base app is a navigation app that supports the offline usage of maps, but users can also download add-ons supporting satellite images [34], augmented reality [35] or geo-caching [36].

In the Mascara threat model, the user just needs to install a single malicious app from the Play Store to become a victim. In fact, the attacker first uses our framework Mascarer to customize the Container App and generate the malicious add-on $V'$ for the specific victim app $V$. Then, the attacker publishes $V'$ on the Play Store as an important update for $V$ and the user, deceived by $V'$ features, installs it on his smartphone. Since users typically do not investigate the apps they download [37], this is a very simple requirement.

In Fig. 3, we illustrate the whole Mascara attack. The first step is publishing the malicious add-on $V'$, generated through Mascarer, on the Google Play Store (step 1). Then, any Android user, already having the victim app $V$ installed on his device, can become a victim by installing $V'$ (step 2). During the activation of $V'$, this downloads a malicious APK from a remote server and executes it in its virtual environment (step 3). At the same time, it creates a new shortcut in the Home screen of the device, that looks like a shortcut of $V$, while it executes $V'$ (step 4). From now on, whenever the user presses that shortcut, $V'$ is executed (step 5) to load $V$ (step 6) and to launch Mascara against the user.

![Fig. 3. Complete workflow of Mascara.](image)

To implement Mascara, we identified the following design goals: (i) the malicious add-on performs the attack against a specific app; (ii) the malicious add-on builds a virtual environment, in which both the victim app and the malicious APK run; (iii) the malicious add-on requires no modification to the victim app source code; (iv) the malicious add-on requires no modification to the underlying Android OS; (v) the malicious add-on is a standard Android app, which can be installed by any user.

B. Architecture of the Malicious Add-On

The architecture of the malicious add-on, which is depicted in Fig. 4, encompasses the following components: the Victim App, the Malicious APK and the Hooking Modules.

The Victim App component refers to the set of classes and resources belonging to the victim app, which is already installed on the device. As a matter of fact, the malicious add-on loads such files directly from the victim app APK, which is world-readable, and executes them in its virtual environment. Together with the Victim App, the malicious add-on runs in its virtual environment also the Malicious APK, which exploits the UID shared with the victim app to harm the final user in several ways (e.g., extracting or modifying sensitive data). At the installation time, the malicious add-on does not contain the Malicious APK, which is instead downloaded at runtime.
from a remote server. Finally, the **Hooking modules** include both the *Dynamic Proxy API Module*, a DroidPlugin built-in module supporting the components lifecycle, and *Whale* [38], an external hooking library. Both the *Dynamic Proxy API Module* and *Whale* are used to install new hooks and bypass existing defence mechanisms [21], [17].

**V. Mascarer FRAMEWORK DESIGN**

In this section, we illustrate how *Mascarer* has been designed to generate a malicious add-on for a given victim app. In particular, *Mascarer* executes the following two procedures: (i) the customization process (Section V-A); (ii) the exploitation of the virtualization technique (Section V-B).

**A. Customization Process**

By default, the malicious add-on inherits the *AndroidManifest.xml* file from DroidPlugin, which declares all existing Android permissions and a set of predefined stub components, having the same repetitive structure. The purpose of the customization process is to make the malicious add-on resilient to the existing defence mechanisms by modifying its: (i) permissions, (ii) components and (iii) resources. More details about the defence mechanisms will be illustrated in Section VIII.

**Permissions.** In addition to the defence mechanisms against virtualization, Android antiviruses [39] could also detect a malicious add-on declaring all Android permissions, since it violates the Android least-privilege security model (i.e., Android apps should declare the minimum amount of permissions required to work properly). Thus, during the customization process, *Mascarer* adjusts both the malicious add-on and the *Malicious APK* *AndroidManifest.xml* files to declare the same permissions as the victim app. This way, the malicious add-on will be able to run the victim app in its virtual environment and the *Malicious APK* will be able to exploit the same permissions, without declaring additional ones.

**Components.** As for the permissions, the malicious add-on first inherits all the stub components declared in the DroidPlugin *AndroidManifest.xml* file. Those stub components are exclusively used to change the names of the victim app’s components at runtime. Thus, they have a very repetitive structure and meaningless names, which help defence mechanisms to detect frameworks supporting the generation of virtual environments. However, *Mascarer* modifies both the malicious add-on and the *Malicious APK* *AndroidManifest.xml* files. Concerning the malicious add-on, *Mascarer* copies all the components of the victim app, preserving their original names, and modifies the additional components by giving them a name correlated with the victim app’s name. With the same approach, *Mascarer* customizes also the *Malicious APK* components.

**Resources.** Since the malicious add-on needs a custom launcher icon, when it is installed on the device, and the victim app launcher icon, to create the shortcut in user Home screen, *Mascarer* copies those resources into the malicious add-on’s ones.

**B. Exploitation of the Virtualization Technique**

*Mascarer* relies on two features of the virtualization technique: (i) the UID shared among all apps running within the same virtual environment; (ii) the hooking mechanism that allows to intercept APIs and change their return values at runtime.

**UID sharing among plugin apps.** In Android, sharing the same UID means also sharing the same permissions and having access to the same private directory. To exploit the shared UID, we rely on the DroidPlugin feature to run multiple plugin apps in the same virtual environment. Thus, our design involves that the malicious add-on runs only the victim app and a single *Malicious APK* within the same virtual environment. Even if multiple apps can run simultaneously in the same virtual environment, only one can run in foreground and the others in background. According to this, we designed the *Malicious APK* as an app with no graphical components (i.e., Activities), but only Services. In particular, by default, the *Malicious APK* declares a set of Services, each one aimed at stealing/injecting specific data into the user smartphone. Some of such Services might need access to protected resources, thus, the *Malicious APK* declares also the required permissions. During the customization process, *Mascarer* modifies the *Malicious APK* by removing all the Services besides the ones that can exploit the victim app’s permissions. Once the *Malicious APK* is downloaded from the remote server, the malicious add-on activates all its Services. Thanks to this design, at runtime the malicious add-on runs the victim app in foreground and all the malicious APK’s Services in background.

**Hooking mechanism.** By design, the *Dynamic Proxy API Module* of DroidPlugin intercepts the outgoing requests and the incoming replies to dynamically change the names of the referenced components and guarantee the lifecycle of plugin apps’ components. However, the same architecture can be used to install new hooks that either bypass existing defence mechanisms or damage the user. To install new hooks we both exploited the DroidPlugin hooking mechanism and the *Whale* library.
VI. Mascarer Framework Implementation

In this section, we provide the technical details of the Mascarer implementation concerning the customization process (Section VI-A), the first execution of the malicious add-on (Section VI-B) and the exploitation of the virtualization technique (Section VI-C).

A. Customization Process

We developed the customization process as a script, which performs the following steps: (i) updating permissions and features of the malicious add-on; (ii) customizing the malicious APK, which will be downloaded at runtime; (iii) updating the components of the malicious add-on; (iv) retrieving the necessary resources from the victim app to be copied into the malicious add-on.

Considering the first step, the customization process relies on the Androguard Python tool [40] to retrieve all permissions and features declared by the victim app and copy them into the malicious add-on AndroidManifest.xml file. As a result, the add-on has the same permissions as the victim app plus the INSTALL_SHORTCUT and the KILL_BACKGROUND_PROCESSES permissions used to create the fake shortcut in the Home screen and to stop the process of the victim app, respectively.

During the second step, the customization process modifies the malicious APK so that it exploits only the permissions of the victim app. Thus, the permissions and associated Services of the malicious APK, which are not declared by the victim app, are removed.

The purpose of the third step is to retrieve all components names of the victim app, as well as of the malicious APK, and create the equivalent components (i.e., names and types) in the AndroidManifest.xml file of the malicious add-on. In addition, since each component declared in the AndroidManiifest.xml file needs a match with a .java file, the script creates the necessary ones in the add-on. Once again, to inspect the victim app and the malicious APK components, the customization process relies on the Androguard tool. Finally, the script customizes also the names of the remaining add-on components, which belong to the DroidPlugin framework and originally come with their default names (e.g., if Telegram is the victim app, the PluginServiceManager becomes TelegramServiceManager).

During the fourth and last step, the script retrieves first the victim app name through Androguard and then its launcher icon from its resources. To achieve the second aim, the script gets first the name of the launcher icon through Androguard, then it unzips the victim app APK file and gets the icon under the predefined path. Since drawable images are not involved in the compilation process of an Android app, they do not need to be decompiled.

As shown in Fig. 5, at the end of the customization process, the add-on encompasses components belonging to the victim app and to the malicious APK, as well as additional malicious code placed in its own logic.

The customization process updates the following components inside the malicious add-on: the res/ folder, the AndroidManifest.xml file, the classes.dex file and the resources.arsc file.

The res/ folder initially contains only the add-on uncompiled resources, among which the launcher icon and the XML layout file of the Activity shown to the user to confirm the add-on activation. In the same folder, the customization process saves the victim app launcher icon used to create the fake shortcut in the Home screen.

After the customization, the AndroidManifest.xml file stores items belonging to the add-on, to the victim app and to the malicious APK as follows: items of the add-on (i.e., the INSTALL_SHORTCUT and KILL_BACKGROUND_PROCESSES permissions, the Activity shown to the user during the first execution, all components belonging to the DroidPlugin framework and required to build a virtual environment); items of the victim app (i.e., all its permissions, components and features); items of the malicious APK (i.e., all its components).

The classes.dex file contains the whole compiled code of an Android app. Considering the add-on, this file encompasses all classes belonging to the DroidPlugin framework plus the malicious hooks of the Dynamic Proxy Module and Whale library to bypass the existing defence mechanisms.

Finally, the resources.arsc file contains all information about resources, XML nodes and their attributes. After the customization, it comprises the add-on resources and the victim app launcher icon.

B. Starting the Malicious Add-on

During its first execution, the add-on performs the following activities: (i) stopping the victim app process already running on the phone through the KILL_BACKGROUND_PROCESSES permission; (ii) exploiting the INSTALL_SHORTCUT permission to create a fake shortcut with the same icon and label of
the victim app, but making it able to launch the malicious add-on; (iii) downloading the malicious APK from a remote server; (iv) executing the malicious APK.

C. Exploitation of the Virtualization Technique

Our design of Mascara involves that, at runtime, the add-on downloads a malicious APK exploiting the most dangerous permissions declared by the victim app (i.e., those allowing to access to or to modify sensitive data available on the phone). According to our design, the add-on runs simultaneously the victim app and the malicious APK in the same virtual environment. However, to prevent the user from noticing the existence of the malicious APK, we designed it as an app declaring only Services, without any graphical components. Thus, each Service aims at exploiting a specific Android permission and the whole set of Services declared by the malicious APK is everytime customized according to the victim app. As a result, at runtime the add-on launches all malicious APK Services and runs them in background, together with the victim app. In our implementation, we developed separate Services for the following permissions: the READ_CONTACTS permission to acquire the contacts already saved on the device; the READ_SMS permission to access the SMS already saved on the device; the RECEIVE_SMS permission for intercepting any new incoming SMS; the READ_PHONE_STATE permission for accessing the state of the phone; the READ_CALL_LOG permission for reading the user’s call logs; the CAMERA permission for randomly taking pictures through the user phone; the RECORD_AUDIO permission for reading the user's call logs; the CALL_LOG permission for accessing the state of the phone; the READ_PHONE_STATE permission to send the collected information to our remote server.

To further exploit the virtualization technique, we introduced new hooks, through DroidPlugin and Whale, by identifying specific Android APIs and modifying their return values to bypass the existing defence mechanisms.

**DroidPlugin hooking mechanism.** If the hook is located in an Android system service, the hooking process requires the following steps:

- Compatibility class (i.e., ExampleCompat): this class has to declare the static methods Class() and asInterface(). The name of the specified Class should be the same as the one of the Interface containing all the methods that need to be hooked.
- Hook handle class (i.e., ExampleHookHandle): it contains a set of inner classes, one for each method that needs to be hooked. Each inner class allows to specify the expected behaviour before and after the invocation of the method;
- Hook binder class (i.e., ExampleBinderHook): this class returns a proxy to the Android system service hooked, since the original one is cached, and an instance of the hook handle;
- Compatibility class (i.e., ExampleCompat): this class has to declare the static methods Class() and asInterface(). The name of the specified Class should be the same as the one of the Interface containing all the methods that need to be hooked.

The hook can be installed through the method installHook() of the HookFactory class.

**Whale hooking mechanism.** To install a hook in a different object other than an Android system service, we can rely on the Whale [38] library. Whale uses the Xposed-style method hooking and it can bypass the Hidden API policy, the restriction applied to third-party apps not signed with the platform signature. Also, Whale can modify the inheritance relationship between classes and the class an object belongs to. Through the Xposed-style method hooking, it is possible to replace the entire method body or introduce new code before and after the original method invocation. When a method is invoked, the execution goes through the injected code and, then, to the original code. The method findAndHookMethod is used to find the target method and the callback can be:

- XC_MethodHook: the class that edits the fields before and after the original method invocation.
- XC_MethodReplacement: the class that replaces the original method with a new one.

VII. Mascara AND Mascarer EVALUATION

In this section, we first analyze the efficiency of Mascarer by measuring the compilation time required to generate the malicious add-ons for 100 popular apps. Then, we evaluate the effectiveness of Mascara by implementing the attack against the Alamo, Amazon Music and Telegram apps and by measuring the overhead introduced at runtime by the virtualization technique. For all experiments, we used a Samsung Galaxy A5 (SM-A5000FU) running Android Marshmallow 6.0.1.
Malicious Add-On Compilation Time. Fig. 7 shows the average compilation time (over ten rounds) required to generate the malicious add-ons for 100 popular apps available on the Google Play Store. To run the experiment, we used an ASUS VivoBook Pro N580GD-E4087T (Processor: Intel® Core™ i7 8750H; Memory: 16GB DDR4 2400 MHz SDRAM; Video card: NVIDIA® GeForce® GTX 1050 4GB DDR5; Storage: SSD M.2 512 GB SATA 3.0 + HDD 2.5” 1 TB 5400 RPM) running Linux Mint 19.3 Cinnamon. The complete list of the analyzed apps is available in Appendix A. The results show that for most of the malicious add-ons the compilation time is less than ten seconds. Considering the estimated amount of time that a smart attacker might need to manually generate the malicious add-on for a single app, this is a threatening result since Mascara allows to build the attack on a large scale set of apps with a limited amount of time.

Virtualization Technique Runtime Overhead. To measure the runtime overhead introduced by the virtualization technique, we used the Java microbenchmark CaffeineMark (version 3.0) [41] ported on the Android platform. This benchmark runs a set of tests which allows a user to assess different aspects of virtual machine performance. The benchmark does not produce absolute values for the tests. Instead, it uses internal scoring measures, which are useful only in case of comparison with other systems. The overall score of CaffeineMark 3.0 is a geometric mean of all individual tests. To assess the virtual environment generated by DroidPlugin, we compared it with Stock Android system (version 6.0.1). The results are shown in Fig. 8. From the figure, we can observe that in the virtual environment the performances are reduced: the higher the score, the faster the system.

VIII. DEFENCE MECHANISMS AGAINST ANDROID VIRTUALIZATION

Among the existing defence mechanisms, we considered the AntiPlugin [21] and DiPrint [17] solutions because their source code is available online and they provide a more comprehensive approach in detecting virtual environments with source code is available online and they provide a more comprehensive approach in detecting virtual environments with respect to other mechanisms [16], [15], [20]. In particular, we experimentally evaluated AntiPlugin and DiPrint by running both of them in Mascara and verifying whether they can detect our virtual environment. In the rest of this section, we deeply describe each detection mechanism used by AntiPlugin and DiPrint and how we managed to bypass it.

A. Feature for Virtual Environment Detection: Permissions

By default, DroidPlugin declares all Android permissions to enable any app to run in its virtual environment.

Detection Mechanism 1. Under VBA, the AntiPlugin App is granted more permissions than expected, since it shares the same UID with the container app. Thus, the AntiPlugin App can access to resources, which it should not be allowed to (e.g., the BLUETOOTH and BLUETOOTH_ADMIN permissions are normal permissions, which are automatically granted at the installation and do not require a confirmation from the user). Since in Mascara the malicious add-on declares the permissions of the victim app, the AntiPlugin App is not able to access to different resources than the expected ones.

Detection Mechanism 2. The AntiPlugin App can check the permissions associated to the app with its own package name through the PackageInfo object [42]. Under VBA, the AntiPlugin App’s request triggers an error, since the app is not installed on the device. In Mascara, the AntiPlugin App is already installed and the request returns the expected list of permissions.

Detection Mechanism 3. DiPrint checks the undeclared permissions by calling the checkCallingOrSelfPermission API. Moreover, DiPrint attempts to perform a set of actions that require dangerous permissions like CAMERA, RECORD_AUDIO and READ_SMS. In Mascara, the malicious add-on declares the same permissions as the victim app, thus, DiPrint is not able to access to different resources than the expected ones.

B. Feature for Virtual Environment Detection: Package Name

Detection Mechanism 4. Under VBA, the AntiPlugin App is not installed on the device, which means that also the Android OS did not register it. Thus, the AntiPlugin App queries the PackageManager [43] to get the list of all installed apps and check whether its package is in that list. Since Mascara requires the victim app to be installed on the device, the AntiPlugin App will receive a positive reply.

C. Feature for Virtual Environment Detection: App component name

Detection Mechanism 5. The getRecentTasks(int, int) [44] and getRunningTasks(int) [45] APIs return the list of tasks recently launched by the user or currently running (a task is a collection of activities that users interact with when performing a certain job). Originally, any app could retrieve the list of tasks, even the ones referring to Activities of other apps. From Android level 21, those APIs were declared as deprecated and restricted to return only the caller’s own tasks, i.e. only the list of Activities belonging to the app that called the API. Under VBA, the AntiPlugin App can detect the tasks of other Plugin Apps running in the same virtual environment. In Mascara, we designed the Malicious APK as an Android app declaring only Services, not Activities. Thus, even if calling the target API, the AntiPlugin App will see only its own tasks.

Detection Mechanism 6. The getRunningServices(int) [46] API returns the list of running Services. Under VBA, the AntiPlugin App might see Services belonging to other components running in the same virtual environment. By executing AntiPlugin in Mascara, we found out that this approach does not work properly.

D. Feature for Virtual Environment Detection: Process Info

Detection Mechanism 7. The getRunningAppProcesses() [47] API returns the list of apps processes running on the device. If called by the AntiPlugin App under a
VBA, this API returns at least the Container App process in addition to the AntiPlugin App’s one. In Mascara, all the processes under the same UID share the same name. However, through the Dynamic Proxy API Module of DroidPlugin, we hooked the getRunningAppProcesses() method so that all process names are replaced with the victim app name.

**Detection Mechanism 8.** DiPrint call the Runtime.exec(String) instruction with the ps argument to retrieve all processes information. In Mascara, we used Whale to hook the Runtime.exec(String) instruction and replace the ps argument with the ls one.

**E. Feature for Virtual Environment Detection: Internal Storage Info**

For each newly installed app, the Android OS creates a private directory with the following path: /data/data/package_name. On the contrary, DroidPlugin assigns each Plugin App a private directory with the following pattern: /data/data/container_package_name/Plugin/plugin_package_name.

**Detection Mechanism 9.** The AntiPlugin App can check whether its private directory follows the default Android pattern. If not, the VBA can be detected. However, in Mascara we hooked the getApplicationInfo API through the Dynamic Proxy API Module of DroidPlugin to return the default Android directory pattern.

**Detection Mechanism 10.** Under VBA, the AntiPlugin App can get the location of its APK by inspecting the sourceDir field of the ApplicationInfo object [48]. Since the Android framework returns the ApplicationInfo for a given package name, in Mascara, the AntiPlugin App will retrieve the one referring to the victim app installed on the mobile device.

**Detection Mechanism 11.** Under VBA, the process memory contains also the malicious APK path. DiPrint searches
the existence of another different APK path in its own process memory by reading "/proc/self/maps". In Mascara, any access to "/proc/self/maps" is disabled by hooking the constructor of the File class through Whale.

**Detection Mechanism 12.** Android loads the dynamic-link library path from "/data/app/package". Under VBA dynamic-link libraries are loaded from a different path than the original. DiPrint searches in the process memory the existence of suspicious paths. However, In Mascara, any access to "/proc/self/maps" is disabled by hooking the constructor of the File class through Whale.

**F. Feature for Virtual Environment Detection: Number of Launched App Activity and Service**

**Detection Mechanism 13.** By default, DroidPlugin declares only one stub Service. Under VBA, the AntiPlugin App can launch a higher number of Services (or even other components) to trigger an exception. In Mascara, the malicious add-on undergoes a customization process, through which all the victim app's components are copied into its AndroidManifest.xml file. Thus, the AntiPlugin App has the exact number and type of expected components.

**Detection Mechanism 14.** The AntiPlugin App can inspect the number and type of components associated to its package name by analyzing the PackageInfo object [42]. In Mascara, the AntiPlugin App will retrieve the PackageInfo object associated to the victim app, already installed on the device.

**G. Feature for Virtual Environment Detection: Static Broadcast Receiver**

**Detection Mechanism 15.** Among its stub components, DroidPlugin does not declare Broadcast Receivers since those are parsed from the AntiPlugin App's AndroidManifest.xml and dynamically registered at runtime. The AntiPlugin App unregisters all the dynamically registered receivers and sends them an event to confirm that the process was successful. In Mascara, all victim app components are copied into the malicious add-on, including Broadcast Receivers.

**H. Feature for Virtual Environment Detection: Change App Component Property at Runtime**

**Detection Mechanism 16.** The AntiPlugin App calls the setComponentEnabledSetting(ComponentName, int, int) [49] API to change the properties of a component at runtime. Under VBA, that call will trigger an error since the component will not be registered. By executing AntiPlugin in Mascara, we found out that this approach does not work properly.

**I. Feature for Virtual Environment Detection: Shared Native Components**

**Detection Mechanism 17.** Under VBA, the AntiPlugin App might share the same virtual environment with other apps. If anyone has native components, such as WebView, these will share some internal data with other apps that have the same components. In Mascara, we designed the malicious add-on so that the Malicious APK does not have native components.

**J. Feature for Virtual Environment Detection: Stack tracking of exception**

**Detection Mechanism 18.** DiPrint analyzes the stack trace of the 13 lifecycle functions for four components. Under VBA, the stack trace should be different. By executing DiPrint in Mascara, we found out that this approach does not work properly.

**IX. Discussion**

In this section, we illustrate the Mascara limitations (Section IX-A) and the new detection mechanism we designed (Section IX-B).

**A. Limitations**

The first limitation consists in the execution of the victim app in a new Context, which is not its original one. When the malicious add-on loads the victim app classes and resources and executes them in the virtual environment, the victim app gains a new UID and loses the access to the data provided by the user until that time. Therefore, at the first execution, the victim app will loose its original state. The second limitation regards the startup time of the victim app, when launched for the first time inside the virtual environment, which might take few seconds (6.64 sec on average for Telegram over ten attempts). Finally, the last limitation involves the creation of the shortcut in the Home screen, which could already exist on the user smartphone.

**B. Countermeasures**

The existing defence mechanisms rely on the analysis of virtual environments properties retrieved through specific APIs, which could be easily modified by a malicious Container App, that leverages on hooking techniques. This is the approach we used with Mascara, thus, proving the ineffectiveness of existing defence mechanisms. On the contrary, to find a novel and robust countermeasure, we inspected the internal data structures of the Android Runtime (ART) [50], the Android apps execution environment. At the installation time, ART applies the Ahead-of-Time (AoT) compilation to compile Dalvik bytecode into binary code. Moreover, ART mirrors Java classes and methods of the Android framework and of the app custom code into C++ classes: the Class class and the ArtMethod class, respectively. At runtime, such classes are queried to retrieve the location of the binary code that needs to be executed. In Android 7.0, ART introduced a hybrid approach, based on both the Just-in-Time (JIT) compilation and the AoT one. Thus, the most frequently used methods are compiled AoT at installation time, while the remaining ones are interpreted. The key component of the hybrid compilation is the hotness_count field of ArtMethod classes. This field provides the number of invocations and of loop iterations for a specific method, which is compiled
according to the hotness\_count value. By analyzing the hotness\_count value for different methods in different Android versions, we found that a hotness\_count equal to 0 corresponds to the Aot. We, thus, chose the hotness\_count value of the currentActivityThread method in the ActivityThread class and evaluate its value when an app runs in the Android native environment and when it runs in a virtual environment. As shown in Fig. 9, the hotness\_count value is 0 for all the apps on the Play Store supporting the virtualization technique, which also means they all use the AoT compilation. Based on this observation, we then developed a robust detection library (i.e., Singular) that detects virtual environments by considering the hotness\_count value of ArtMethod classes.

![Comparison of hotness\_count values in Android native and virtual environments.](image)

### X. Conclusions

In this paper, we proposed Mascara, a novel attack which relies on the virtualization technique to access and modify sensitive data available on the user’s phone, as well as runtime data provided by the user. Mascara involves a malicious add-on, which builds a virtual environment and runs both the victim app and a malicious APK inside it, gaining full control of the first one at runtime. We developed Mascarer, a framework that automatically generates the malicious add-ons, and we tested it over 100 apps. We analyzed the runtime overhead introduced by the virtual environment and built up the Mascara attack against Alamo, Amazon Music and Telegram. We, finally, identified a robust approach to determine whether an app is running in a virtual environment, by inspecting the hotness\_count value for ArtMethod classes in ART.

### References

[1] AppBrain. “Number of android applications,” 2020. [Online]. Available: https://www.appbrain.com/stats/stats\_index

[2] “Google bouncer,” 2020, last access June 2020. [Online]. Available: https://blog.trendmicro.com/trendlabs\_security\_intelligence/a\_look\_at\_google\_bouncer/

[3] “Google play protect,” 2020, last access June 2020. [Online]. Available: https://support.google.com/play\_apps\_answer/2812853

[4] “Mobile threat statistics,” 2019, last access June 2020. [Online]. Available: https://securelist.com/it\_threat\_evolution\_q3\_2019\_statistics/95269/

[5] G. Developers, 2020. [Online]. Available: https://developer.android.com/ studio/build/multidex

[6] V. Costamagna and C. Zheng, “Artroid: A virtual-method hooking framework on android ART runtime,” in Proceedings of the 1st International Workshop on Innovations in Mobile Privacy and Security, IMPS 2016, co-located with the International Symposium on Engineering Secure Software and Systems (ESSoS 2016), London, UK, April 6, 2016., 2016, pp. 20–28. [Online]. Available: http://ceur-ws.org/Vol-1575/paper\_10.pdf

[7] J. Bao, Android App-Hook and Plug-In Technology. CRC Press, 2020.

[8] “Improve the security of android* applications using hooking techniques,” 2016, last access June 2020. [Online]. Available: https://software.intel.com/en-us/articles/improve-the-security-of-android-applications-using-hooking-techniques-part-1

[9] “Parallel space - fmulti account,” 2020, last access June 2020. [Online]. Available: https://play.google.com/store/apps\_details?id=com.lbe.parallel

[10] “Multiple accounts; parallel app,” 2020, last access June 2020. [Online]. Available: https://play.google.com/store/apps\_details?id=com.excellence.multiaccounts

[11] Google, “Supporting multiple users,” 2020. [Online]. Available: https://source.android.com/devices/tech/admin/multi-users

[12] 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, ser. RESEC ’18. New York, NY, USA: ACM, 2018, pp. 61–64. [Online]. Available: http://doi.acm.org/10.1145/3203422.3203425

[13] C. Xuany, G. Cheng, and E. Stuntebeck, “Droidpill: Pwn your daily-use apps,” in Proceedings of the 12th ACM Symposium on Information, Computer and Communications Security, April 2017.

[14] L. Zhang, Z. Yang, Y. He, M. Li, S. Yang, M. Yang, Y. Zhang, and Z. Qian, “In the middle: Demystify application virtualization in android and its security threats,” Proc. ACM Meas. Anal. Comput. Syst., vol. 3, no. 1, Mar. 2019. [Online]. Available: https://doi.org/10.1145/3322205.3311088

[15] D. Dai, R. Li, J. Tang, A. Dervanian, 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, ser. SACMAT ’15. New York, NY, USA: Association for Computing Machinery, 2020, p. 25–32. [Online]. Available: https://doi.org/10.1145/3319911.3395608

[16] 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, ser. MobiSys ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 222–235. [Online]. Available: https://doi.org/10.1145/3307334.3326072

[17] Y. Wu, J. Huang, B. Liang, and W. Shi, “Do not jail my app: Detecting the android plugin environments by time lag contradiction,” Journal of Computer Security, vol. 28, pp. 1–25, 01 2020.

[18] “Malware posing as dual instance app steals users’ twitter credentials,” 2016, last access June 2020. [Online]. Available: https://blog.avast.com/ malware-posing-as-dual-instance-app-steals-users-twitter-credentials

[19] “Fake whatsapp app downloaded more than one million times,” 2017, last access June 2020. [Online]. Available: https://www.bbc.com/news/ technology-41886157

[20] “Traveling: A security analysis of app-level virtualization in android,” in Proceedings of the 25th ACM Symposium on Access Control Models and Technologies, ser. SACMAT ’15. New York, NY, USA: Association for Computing Machinery, 2020, p. 25–32. [Online]. Available: https://doi.org/10.1145/3319911.3395608

[21] T. Luo, C. Zheng, Z. Xu, and X. Ouyang, “Anti-plugin: Don’t let your app play as an android plugin,” in Proceedings of Blackhat Asia 2017, 2017.

[22] “Droidplugin,” 2020, last access June 2020. [Online]. Available: https://github.com/DroidPluginTeam/DroidPlugin

[23] “Virtualapp,” 2020, last access June 2020. [Online]. Available: https://github.com/AdLiDY/ VirtualApp

[24] “Dynamicapk,” 2020, last access June 2020. [Online]. Available: https://github.com/CripMobile/DynamicAPK

[25] “Dynamic proxy classes,” 2020, last access June 2020. [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/ reflection/proxy.html

[26] A. Bianchi, Y. Fratantonio, C. Kruegel, and G. Vigna, “Njas: Sandboxing unmodified applications in non-rooted devices running stock android,” pp. 27–38, 2015. [Online]. Available: http://doi.acm. org/10.1145/2808117.2808122

[27] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. von Styp-Rekowsky, “Bofix: Full-fledged app sandboxing for stock android,” in 24th USENIX Security Symposium (USENIX Security 15). Washington, D.C.: USENIX Association, 2015, pp. 691–706. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/backes
[28] V. M. Afonso, P. L. de Geus, A. Bianchi, Y. Fratantonio, C. Krügel, G. Vigna, A. Doupé, and M. Polino, “Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy,” in NDSS, 2016.

[29] B. Davis, B. S. A. Khodaverdian, and H. Chen, “I-arm-droid: A rewriting framework for in-app reference monitors for android applications,” in In Proceedings of the Mobile Security Technologies 2012, MOST ’12. IEEE, 2012.

[30] Y. Zhou, K. Patel, L. Wu, Z. Wang, and X. Jiang, “Hybrid user-level sandboxing of third-party android apps,” in Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ser. ASIA CCS ’15. New York, NY, USA: Association for Computing Machinery, 2015, p. 19–30. [Online]. Available: https://doi.org/10.1145/2714576.2714598

[31] R. Xu, H. Sáidi, and R. Anderson, “Aurasium: Practical policy enforcement for android applications;” in Proceedings of the 21st USENIX Conference on Security Symposium, ser. Security’12. USA: USENIX Association, 2012, p. 27.

[32] 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 NDSS, 2018.

[33] “Locus map free,” 2020, last access June 2020. [Online]. Available: https://play.google.com/store/apps/details?id=menion.android.locus

[34] “Caffeinemark 3.0 benchmark,” 1997, last access June 2020. [Online]. Available: http://www.benchmarkhq.ru/cm30/

[35] “Locus-addon satellite images,” 2020, last access June 2020. [Online]. Available: https://play.google.com/store/apps/details?id=cz.jerabekjakub.locusaddon.offlinegooglemaps

[36] “Augmented reality for locus map,” 2020, last access June 2020. [Online]. Available: https://play.google.com/store/apps/details?id=com.arcao.geocaching4locus

[37] “Fake mobile apps, a growing threat,” 2019, last access June 2020. [Online]. Available: https://www.guardsquare.com/en/blog/fake-mobile-apps-growing-threat-2019

[38] Whale, 2020. [Online]. Available: https://github.com/usLody/whale

[39] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, P. G. Bringas, and G. Álvarez, “PUMA: Permission Usage to Detect Malware in Android,” in International Joint Conference CSIIS’12-ICEUTE’12-SOCO’12 Special Sessions, Á. Herrero, V. Sáácel, A. Abraham, I. Zelinka, B. Baraque, H. Quintián, J. L. Calvo, J. Sedano, and E. Corchado, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 289–298.

[40] Androguard, “Androguard’s documentation,” 2020. [Online]. Available: https://androguard.readthedocs.io/en/latest/

[41] “Package manager’s documentation,” 2020. [Online]. Available: https://developer.android.com/reference/android/content/pm/ApplicationInfo

[42] “Activity Manager’s documentation,” 2020. [Online]. Available: https://developer.android.com/reference/android/app/ActivityManager.html#setComponentEnabledSetting(android.content.ComponentName,int)
| Index | App Name               | Package Name                                      | Size (MB) | N. Download     |
|-------|------------------------|---------------------------------------------------|-----------|-----------------|
| 1     | 8 Ball Pool            | com.miniclip.eightballpool                        | 13.86     | 500,000,000+    |
| 2     | Alibaba.com            | com.alibaba.intl.android.apps.poseidon            | 16.16     | 100,000,000+    |
| 3     | Amazon Music           | com.amazon.mp3                                    | 22.45     | 100,000,000+    |
| 4     | Amazon Photos          | com.amazon.cloudrive.photos                       | 9.62      | 10,000,000+     |
| 5     | Amazon Prime Video     | com.amazon.avod.thirdpartyclient                 | 39.56     | 100,000,000+    |
| 6     | Amazon Shopping        | com.amazon.mShop.android.shopping                | 2.97      | 100,000,000+    |
| 7     | Aquapark.io            | com.cassette.aquapark                             | 11.68     | 100,000,000+    |
| 8     | Art of War: Legions    | com.addictive.strategy.army                      | 29.43     | 10,000,000+     |
| 9     | Audible Audiobooks     | com.audible.application                           | 31.51     | 100,000,000+    |
| 10    | Ball Pass 3D           | com.kjoeb.BallPass                                | 13.3      | 1,000,000+      |
| 11    | BIGO Live              | sg.bigo.live                                      | 31.6      | 100,000,000+    |
| 12    | Border patrol          | com.fivebits.borderpatrol                        | 11.71     | 10,000,000+     |
| 13    | Brain Out              | com.mind.quiz.brain.out                           | 10.07     | 100,000,000+    |
| 14    | Brain Test             | com.unicostudio.braintest                         | 4.77      | 50,000,000+     |
| 15    | Burger King APP        | com.emn8.mobilem8.nativeapp.bk                    | 9.5       | 1,000,000+      |
| 16    | Calm – Meditate,Sleep,Relax | com.calm.android                      | 26.81     | 10,000,000+    |
| 17    | Candy Crush Saga       | com.king.candycrushsaga                          | 7.35      | 1,000,000,000+  |
| 18    | Candy Crush Soda Saga  | com.king.candycrushsodasaga                      | 7.62      | 100,000,000+    |
| 19    | Chick-fil-A            | com.chickfila.cafalflagship                      | 15.87     | 10,000,000+     |
| 20    | Cisco Webex Meetings   | com.cisco.webex.meetings                          | 16.23     | 10,000,000+     |
| 21    | Coin Master            | com.moonactive.coinmaster                        | 11.44     | 100,000,000+    |
| 22    | Cut and Paint          | com.painter.cutandpaint                          | 18.9      | 10,000,000+     |
| 23    | Dancin Road            | com.amanotes.pamadancingroad                    | 14.49     | 50,000,000+     |
| 24    | Dentist Bling          | com.crazylabs.dentist                            | 24.26     | 10,000,000+     |
| 25    | Dig This!              | se.raketspel.digaround                           | 24.22     | 10,000,000+     |
| 26    | Discord                | com.discord                                      | 23.01     | 100,000,000+    |
| 27    | Draw Climber           | com.appadvisory.drawclimber                      | 23.04     | 50,000,000+     |
| 28    | Emoji Home             | com.home.emoticon.emoji                          | 27.82     | 1,000,000+      |
| 29    | Enlight Pixaloop       | com.lighttricks.pixaloop                         | 12.8      | 10,000,000+     |
| 30    | Epic Race 3D           | com.gym.racemame                                 | 18.91     | 10,000,000+     |
| 31    | Extreme Car Driving Simulator | com.alim.racing                           | 11.94     | 100,000,000+    |
| 32    | Funimation             | com.Funimation.FunimationNow.androidtv           | 17.44     | 50,000+         |
| 33    | Gardenscapes           | com.playrix.gardenscapes                         | 11.92     | 100,000,000+    |
| 34    | GrubHub                | com.grubhub.android                              | 21.18     | 10,000,000+     |
| 35    | HBO Max                | com.hbo.hbonow                                   | 8.11      | 10,000,000+     |
| 36    | Hide ‘N Seek 3D        | helperapp.helpertinidenseekhelper               | 7.15      | 5,000,000+      |
| 37    | Hitmasters             | com.playgendary.hitmasters                      | 20.23     | 10,000,000+     |
| 38    | Hole.io                | io.voodoo.holeio                                 | 14.62     | 50,000,000+     |
| 39    | Home Restoration       | com.panteon.homeswethome                        | 14.42     | 5,000,000+      |
| 40    | Homecapes              | com.playrix.homescapes                          | 9.74      | 100,000,000+    |
| 41    | Houseparty             | com.herzick.houseparty                          | 23.19     | 10,000,000+     |
| 42    | Hula hoop              | com.hulahoop.android                            | 18.17     | 10,000,000+     |
| 43    | Hungry Shark Evolution | com.fgol.HungrySharkEvolution                  | 5.21      | 100,000,000+    |
| 44    | I, The One – Action Fighting | vh.one                                     | 26.93     | 10,000,000+    |
| 45    | ID Please – Club Simulation | com.NeverEndingGames.IdPlease                | 25.51     | 10,000,000+    |
| 46    | iHeartRadio            | com.clearchannel.iheartradio.controller          | 47.05     | 50,000,000+     |
| 47    | Johnny Trigger         | com.time.trigger                                 | 14.82     | 50,000,000+     |
| 48    | KeepClean lite         | com.appsinnova.android.keepclean_lite           | 16.52     | 100,000+        |
| 49    | Klondike Adventures    | com.vizorapps.klondike                          | 22.61     | 10,000,000+     |
| 50    | Likee lite             | video.like.lite                                  | 15.43     | 50,000,000+     |
| Index | App Name                | Package Name                      | Size (MB) | N. Download       |
|-------|-------------------------|-----------------------------------|-----------|-------------------|
| 51    | Little Caesars          | com.littlecaesars                 | 10.27     | 5,000,000+        |
| 52    | Microsoft Teams         | com.microsoft.teams               | 21.98     | 50,000,000+       |
| 53    | Mr Ninja                | com.lionstudios.mrninja           | 24.51     | 10,000,000+       |
| 54    | MyBoost                 | com.boost.care                    | 15.62     | 1,000,000+        |
| 55    | NERF Epic Pranks!       | games.nerf.epic.pranks.free       | 3.36      | 10,000,000+       |
| 56    | News Break              | com.particlenews.newsbreak        | 11.75     | 10,000,000+       |
| 57    | News Home               | com.home.news.breaking            | 38.19     | 1,000,000+        |
| 58    | Norton Security VPN     | com.symantec.securewif           | 16.22     | 10,000,000+       |
| 59    | One Booster             | com.cleansteam.oneboost          | 17.5      | 10,000,000+       |
| 60    | Onnect - Pair Matching Puzzle | com.gamebility.onet              | 16.8      | 10,000,000+       |
| 61    | Padenatef               | com.sun.newjbq.beijing.ten       | 28.26     | 500,000+          |
| 62    | Paint By Number         | paint.by.number.pixel.art.coloringdrawing.puzzle | 15.16     | 50,000,000+       |
| 63    | PictureThis             | cn.danatech.xingseus             | 26.36     | 5,000,000+        |
| 64    | Pinterest Lite          | com.pinterest.twa                 | 2.97      | 1,000,000+        |
| 65    | Pizzaiolo!              | com.pizza.dough                  | 17.96     | 1,000,000+        |
| 66    | Postmates Merchant App  | com.postmates.android.merchant   | 24.38     | 100,000+          |
| 67    | Recipes Home            | com.home.recipes.food.drink      | 27.94     | 1,000,000+        |
| 68    | Rescue Cut              | com.app.rescuecut                | 13.57     | 50,000,000+       |
| 69    | Roblox                   | com.robrox.client               | 24.33     | 100,000,000+      |
| 70    | Roku Remote Control     | com.tinybyteapps_robyte          | 7.37      | 5,000,000+        |
| 71    | Shein                   | com.zzkko                         | 33.96     | 100,000,000+      |
| 72    | SiriusXM                 | com.sirius                        | 10.52     | 10,000,000+       |
| 73    | Skype                    | com.skype.raider                 | 9.29      | 1,000,000,000+    |
| 74    | Slap Kings               | mobi.gameguru.slapkings          | 11.56     | 50,000,000+       |
| 75    | SmartNews                | jp.gocro.smartnews.android       | 18.39     | 10,000,000+       |
| 76    | Soap Cutting            | com.crazylabs.soap.cutting       | 23.84     | 10,000,000+       |
| 77    | SONIC Drive-In           | com.sonic.sonicdrivein           | 72.37     | 5,000,000+        |
| 78    | Sort it 3D               | com.game.sortit3d               | 15.05     | 10,000,000+       |
| 79    | Spotify                  | com.spotify.music                | 26.48     | 500,000,000+      |
| 80    | Super Salon              | com.flashread1c.pluckinawesome  | 22.95     | 5,000,000+        |
| 81    | TBN: Watch TV           | tbn_mobile.android               | 23.54     | 500,000+          |
| 82    | The Chosen               | com.vidangel.thechosen           | 22.96     | 1,000,000+        |
| 83    | The Cook                 | com.pd.thecook                   | 14.76     | 10,000,000+       |
| 84    | Tiles Hop                | com.amanotes.beathopper          | 23.57     | 100,000,000+      |
| 85    | Tubi TV                  | com.tubitv                       | 19.37     | 50,000,000+       |
| 86    | Ultimate Disc            | com.marvo.frisbee                | 16.46     | 10,000,000+       |
| 87    | US breaking news         | ma.safe.bnus                     | 13.9      | 1,000,000+        |
| 88    | Video Editor & Video Maker | com.camerasideas.instashot     | 13.33     | 100,000,000+      |
| 89    | Walmart                  | com.walmart.android              | 17.58     | 50,000,000+       |
| 90    | Walmart Grocery          | com.walmart.grocery              | 30.22     | 50,000,000+       |
| 91    | Water Race 3D            | com.ihd.waterrace                | 15.86     | 1,000,000+        |
| 92    | Webtoon                  | com.naver.linewebtoon            | 13.16     | 50,000,000+       |
| 93    | Whatsapp Messenger       | com.whatsapp                     | 11.56     | 5,000,000,000+    |
| 94    | Wish                     | com.contextlogic.wish            | 7.2       | 100,000,000+      |
| 95    | Wood Block Puzzle        | com.fastfun.tetris               | 17.48     | 1,000,000+        |
| 96    | Woodturning              | com.BallGames.Woodturning        | 23.94     | 50,000,000+       |
| 97    | Word Crush               | com.tangramgames.gourddoll.wordcrush | 13.74     | 1,000,000+        |
| 98    | Wordscapes               | com.peoplefun.wordcross          | 17.22     | 10,000,000+       |
| 99    | Yahoo Mail               | com.yahoo.mobile.client.android.mail | 26.51     | 100,000,000+      |
| 100   | Your Phone Companion     | com.microsoft.appmanager         | 24.96     | 100,000,000+      |