

---

# **Malware Analysis on Android**

## **Análisis de Malware en Android**

### **Final Degree Project in Computer Science Degree**

Faculty of Computer Science & Engineering  
Universidad Complutense de Madrid

Course 2020-2021

---



#### **Authors**

Daniel Puente Arribas  
José Ignacio Daguerre Garrido  
Ramón Costales de Ledesma

**Advisors:** Marcos Sánchez-Élez Martín and Inmaculada Pardines Lence



# Abstract

In the XXI century, the world has witnessed the creation, development and proliferation of mobile devices until the massive usage apparent nowadays. The portability, instantaneity and ease of use that these devices offer has encouraged the great majority of the population to have one of them at arm's length. Thus, these devices have become a coveted target for malicious developers. This is the reason why the security of mobile devices has become a vital topic that must be addressed, since a suitable solution has yet to be found.

From this necessity arises the present work, in which we elaborate the beginning of a response that serves as a starting point to promote further development that achieves the desired objective.

With Android being the most representative Operating System among mobile devices, we are going to study the analysis of malware on Android and develop a static and dynamic antivirus based on signatures, permissions and logs, since they will prove useful when trying to detect malicious applications.

## **Keywords (10 max)**

Android, Malware, Vulnerability, Antivirus, Hash, Log, Permission, Static Analysis, Dynamic Analysis, Cloud.



# Resumen

En el siglo XXI se ha podido apreciar la aparición, desarrollo y proliferación de los dispositivos móviles hasta llegar a la masificación que tiene lugar en la actualidad. La portabilidad, instantaneidad y facilidad de uso que ofrecen ha hecho que la mayoría de la población tenga uno siempre al alcance de su mano. Es por ello que se han convertido en un objetivo codiciado por los desarrolladores de programas maliciosos. Así pues, la seguridad de estos dispositivos se ha convertido en un punto clave que debe ser abordado, ya que hasta la fecha no se ha encontrado una solución apropiada.

De esta necesidad surge el presente trabajo, en el que elaboramos el comienzo de una respuesta que sirve como punto de partida para fomentar un posterior desarrollo que alcance el objetivo deseado.

Siendo Android el sistema operativo más representativo entre los dispositivos móviles, vamos a hacer un estudio del análisis del malware en Android y a desarrollar un antivirus estático y dinámico basado en firmas, permisos y logs, pues estas evidencias serán de gran ayuda en la labor de detección de aplicaciones maliciosas.

## ***Palabras clave (máx 10)***

Android, Malware, Vulnerabilidad, Antivirus, Hash, Log, Permiso, Análisis estático, Análisis dinámico, Cloud.



# Acknowledgements

We would like to express our most sincere gratitude to both our project advisors, *Marcos Sánchez-Élez Martín* and *Inmaculada Pardines Lence*, for their excellent guidance throughout the production of this project.

# Index

<b>1. Introduction</b>	<b>11</b>
1.1 Motivations	11
1.2 State of the Art	12
1.3 Objectives	14
1.4 Work planning	14
1.5 Document Organization	21
<b>2. Android</b>	<b>22</b>
2.1 Android Operating System	22
2.1.1 Introduction	22
2.1.2 Applications	24
2.1.3 Components	27
2.1.4 Intents	28
2.2 Android's Vulnerabilities	28
2.2.1 General Vulnerabilities	28
2.2.2 Rooting	31
2.3 Android's Security Systems	33
2.3.1 Linux Security	33
2.3.2 Application Sandbox	33
2.3.3 SELinux	34
2.3.4 System's Security	34
2.3.5 User's Security	35
2.3.6 Apps Security	35
2.4 Android's Malware	35
2.4.1 Malware trends	35
2.4.2 Malware types, aims and characteristics	38
2.4.3 Transmission methods, detection and prevention	42
<b>3. Workspace</b>	<b>46</b>
3.1 Tools and samples	46
3.1.1 Tools	46
3.1.1 Samples	48
3.2 Decisions	49
<b>4. Analysis Methods</b>	<b>51</b>
4.1 Cryptographic Signature Analysis	51
4.2 Heuristic Log Analysis	53
4.2.1 Android logs	53
4.2.2 Log handling	55
4.3 Permission Analysis	56
<b>5. Analysis Methods Implementation</b>	<b>59</b>
5.1 Cryptographic Signature Analysis Implementation	59

5.2 Heuristic Log Analysis Implementation	65
5.2.1 Application	65
5.2.2 Server	79
5.2.3 Problems overcome	84
5.3 Permission Analysis Implementation	85
5.3.1 Dataset of permissions	85
5.3.2 Classification of permissions	89
5.3.3 Permissions analysis and scoring	90
5.3.4 Results	96
<b>6. Android Malware Analyzer App</b>	<b>101</b>
6.1 Fragment Menu	101
6.2 Home Fragment	103
6.3 Apps Information Fragment	104
6.4 Signature Analyzer Fragment	109
6.5 Permission Analyzer Fragment	111
6.6 Log Analyzer Fragment	113
6.7 Previous Results Fragment	118
6.8 Server Settings Fragment	119
6.9 About Us Fragment	122
<b>7. Experimental results</b>	<b>123</b>
7.1 Applications Analyzed	123
7.2 Permission Analysis	132
7.3 Cryptographic Signature Analysis	138
7.4 Heuristic Log Analysis	144
<b>8. Individual Work</b>	<b>152</b>
8.1 Daniel Puente Arribas	152
8.2 José Ignacio Daguerre Garrido	153
8.3 Ramón Costales de Ledesma	154
<b>9. Conclusions &amp; Future Work</b>	<b>155</b>
9.1 Overview	155
9.2 Applied Knowledge	155
9.3 Improvements	156
9.3.1 Signature analyzer	156
9.3.2 Log analyzer	157
9.3.3 Permission analyzer	157
9.4 Future Work	157
9.4.1 Signature analyzer	157
9.4.2 Log analyzer	157
9.4.3 Permission analyzer	158
<b>Bibliography</b>	<b>159</b>



# 1. Introduction

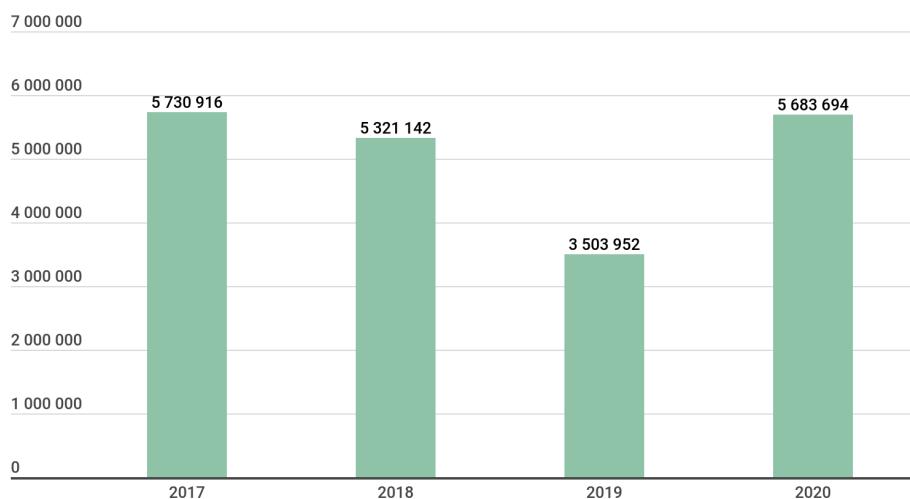
This chapter presents the reasons that have led us to carry out this work, for which it briefly explains previous related works centered on malware analysis. Next, it clarifies the aim of this project. It also provides the path the team has followed in carrying out this work. Finally, it synthesizes the structure of this document.

## 1.1 Motivations

We live in a time in which the proliferation of mobile devices is progressively increasing, thus leading to the estimate that currently 70% of the world's population uses them. Everyone from children to seniors uses mobile devices in different ways, depending on their preferences, hobbies, or business purposes.

More than 2,500 million users choose to use devices with the Android operating system [1], which translates into an 85.9% share in the global mobile market [2]. Therefore, our work has focused on Android as it is the most widely distributed mobile operating system in the world [3].

There is currently a long list of antivirus products that any user can use to protect and prevent attacks on their device [4]. However, most anti-malware has been developed with Windows or Mac OS operating systems in mind, leaving Android on the sidelines. Let us point out that the apparent lack of concern for its security for Android is ironic. There are many Android apps that evade security mechanisms such as Google Play Protect's analysis before being put on the Play Store, and others that are not initially malicious, but then receive an update that changes everything. The increase of malicious applications on Android devices appearing both inside and outside the Play Store itself, together with new attack techniques aimed at mobile devices (SIM Swapping, Smishing, etc.) demonstrate the need for more work in Android security. *Figure 1* shows the number of malicious installation packages on Android mobile devices.



[5] *Figure 1: Number of Android mobile malware installation packages, 2017-2020*

Therefore, the motivation for this project is to study the potential and effectiveness of certain malware analysis techniques adapted to the Android operating system. In addition to the benefit of combining these different techniques in a single application.

## 1.2 State of the Art

First of all we analyze other works related with malware analysis for Android. In particular, we have studied other tools that have been developed previously to solve the same problem; here, we present the more relevant ones.

In the topic of malware analyzers for Android we find several examples:

- **Static analysis:** It consists of analyzing the assembler code without executing the associated binary. It is worth highlighting tools such as DroidMat [6], DREBIN [7], TrustDroid [8] and Androguard [9]. In the latter tool, it analyzes Android files such as resources, XML files and even APKs. Concerning our project, we are dealing with a static analysis focused solely on the comparison of digital signatures (hashes) and we also carry out this analysis based on the permissions of each application installed on an Android device.
- **Dynamic analysis:** It consists of studying the actions of the binary during its execution. Andromaly [10], MADAM [11] (monitors Android at kernel-level and user-level), CrowDroid [12] (compares data behaviour with two data sets based on artificial malware and malware found through machine learning algorithms/techniques) y VirusMeter [13] (checks battery consumption). We make use of a dynamic analysis based on log monitoring.
- **Mixed analysis:** It consists of performing both static and dynamic analysis on Android such as AASandbox [14] or ProfileDroid [15] which scans the software for malicious patterns without installing it (static analysis) and executes the application in a fully isolated environment (dynamic analysis).
- **Metadata analysis:** It consists of collecting and analyzing metadata to detect malware, an example would be the tool WHYPER [16], which identifies app descriptions that involve a specific device permission.

Although we discovered its existence later in this project, the next tool closely resembles the idea of a malware detection application that we wanted to develop in this work.

INCIBE (Instituto Nacional de Ciberseguridad) participated in the development of CONAN mobile [17], an application that helps protect Android mobile devices from malicious applications. It detects if there are any malicious apps installed on your device, and checks if the applications installed are correctly updated and if the configuration of your device is correct by:

- **Configuration:** Analysis of Android OS configurations that may suppose a security risk.
- **Applications:** Analysis of the applications installed on the device.

- **Permisses:** Classification of the permissions that are declared in the applications in the most relevant permissions and by risk category.
- **Proactive service:** It alerts of anomalous and potentially malicious actions.

Finally, in order to understand what an anti-malware tool must deal with, next there is a brief research where it is explained how some of the noted malware mobile attacks of the last year work.

- **System Update Spyware:** On March 26, 2021, Zimperium zLabs warned Android users about a sophisticated new malicious app. The zLabs researchers disclosed unsecured cloud configurations exposing information in thousands of legitimate iOS and Android apps. This new malware disguises itself as a System Update application, and is stealing data, messages, images and taking control of Android phones. Once in control, hackers can record audio and phone calls, take photos, review browser history, access WhatsApp messages, and more [18].
- **Joker:** On January 9, 2020, Google warned the Android users on its security blog about a new threat known as Joker or Bread. It is a malware that has been around since 2017 and tries to bypass all Play Protect protections. Joker has now returned, it is a very active malware which forces Google to be alert. In fact, a total of 1,700 applications containing this malware were removed from the Play Store before they were downloaded by users. However, it is not ruled out that there are more infected applications in the store [19].
- **SIM Swapping:** This is a type of fraud carried out using the SIM card. Such fraud allows access to third-party accounts by exploiting a weakness in multi-factor authentication in which the second factor is a text message (SMS) or a call made to a mobile phone. The fraud exploits the ability of a mobile phone service provider to transfer a phone number to a device that contains a different SIM card. This feature is typically used when a customer has lost or stolen their phone, or is switching service to a new phone. Even though this is not malware that attacks the device directly, usually, the scam starts with a fraudster collecting personal data about the victim, either by using an infected app, phishing emails, buying them from organized criminals, or directly manipulating the victim.

There have been several high-profile attacks using SIM Swapping, including some on social media sites Instagram and Twitter. In 2019, Twitter CEO Jack Dorsey's account was hacked through this method. In May 2020, a lawsuit was filed against 15-year-old Ellis Pinsky from Irvington, New York. He was accused along with twenty other conspirators of defrauding \$ 23.8 million of digital currency investor Michael Terpin, founder and CEO of Transform Group, by using stolen data from smartphones through SIM duplication [20].

## 1.3 Objectives

In this section we proceed to establish the main objectives of this project:

- To introduce the reader to the field of android mobile devices and the malware developed for them.
- To assess the current state of mobile devices and malware trends on Android.
- To briefly describe previously developed malware detection tools for Android devices.
- To convey the importance of having anti-malware software, not only for computers, but also for mobile devices.
- To raise awareness in society of the importance of being cautious when granting permissions to applications.
- To demonstrate the effectiveness of using log analysis on Android.
- To develop an application that can actually be used for detecting possible malware.
- To obtain a clear and simple interface for the application so that the user does not have problems when navigating it.

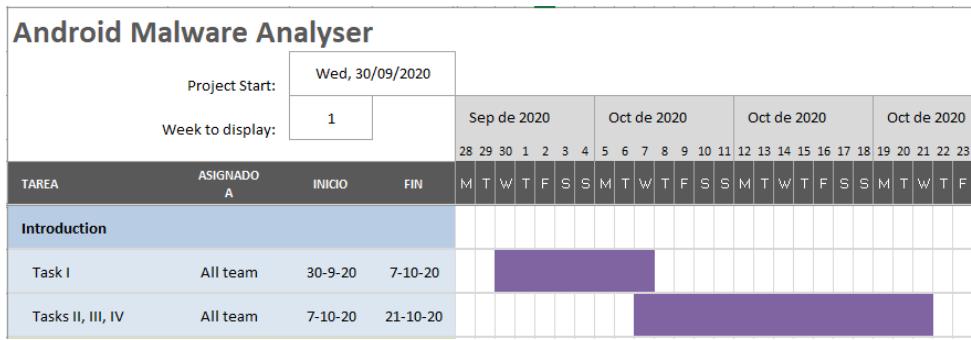
## 1.4 Work planning

When planning our work to achieve the objectives we have divided it into a set of basic tasks and subtasks that we list in this Section. Thus, we also present its temporal development so that the reader can get an idea of the work carried out.

We provide a description for each task, as well as a timestamp and the person who worked on the specific task. In particular, for the case of the development of the app associated with this work, we have decided to differentiate general aspects of its development, named "App", and specific aspects that are named "App [ \* fragment]" with the specific name of the associated task related with the application development.

### 1. Introduction (see *Figure 2*):

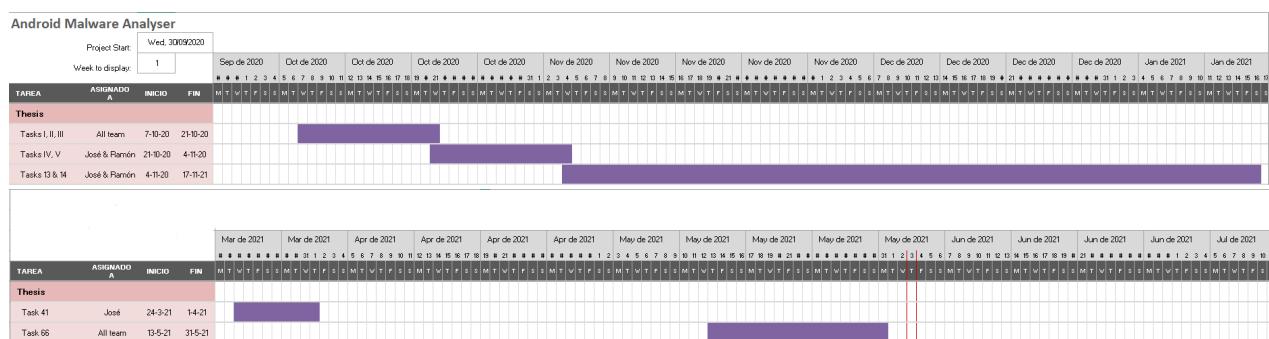
- i. Read similar documents and projects to gather information for our project. Timestamp: 30 September - 7 October. Developers: All the team.
- ii. Establish the workspace, tools. Timestamp: 7 - 21 October. Developers: All the team.
- iii. Establish the scope and objectives of the application. Timestamp: 7 - 21 October. Developers: All the team.
- iv. Search for malware signature databases and malware samples. Timestamp: 7 - 21 October. Developers: All the team.



*Figure 2: Introduction planning*

## 2. Document (see *Figure 3*):

- i. Set the structure of the introduction. Timestamp: 7 - 21 October. Developers: All the team.
  - ii. Search for information to write the Abstract. Timestamp: 7 - 21 October. Developers: All the team.
  - iii. Write the Abstract. Timestamp: 7 - 21 October. Developers: All the team.
  - iv. Search for information on Android Malware. Timestamp: 21 October - 4 November. Developers: José Ignacio Daguerre and Ramón Costales.
  - v. Write the Malware Trends section. Timestamp: 21 October - 4 November. Developers: José Ignacio Daguerre and Ramón Costales.
  - vi. Search for information on Android vulnerabilities. Timestamp: 4 -17 November. Developers: José Ignacio Daguerre and Ramón Costales.
  - vii. Write the General Vulnerabilities section. Timestamp: 4 -17 November. Developers: José Ignacio Daguerre and Ramón Costales.
  - viii. Develop the document. Timestamp: 24 March - 10 April. Developer: José Ignacio Daguerre.
  - ix. Finishing writing the document and structure. Timestamp: 13 - 31 May. Developers: All the team.



*Figure 3: Document planning*

### 3. App (see *Figure 4*):

- i. Develop an application that gets all the installed applications. Timestamp: 7 - 21 October. Developer: Daniel Puente.

- ii. Upgrade the app to list the hashes of all the applications. Timestamp: 21 October - 4 November. Developer: Daniel Puente.
- iii. Upgrade the app to list the permissions of all the applications. Timestamp: 21 October - 4 November. Developer: Daniel Puente.
- iv. Develop the app details activity functionality and view. Timestamp: 4 - 30 November. Developer: Daniel Puente.
- v. Upgrade the app to access the logs. Timestamp: 25 November - 16 December. Developers: Daniel Puente and Ramón Costales.
- vi. Create a mockup local database for the app. Timestamp: 25 November - 16 December. Developer: José Ignacio Daguerre.
- vii. Upgrade the app to connect to the server. Timestamp: 20 January - 8 February. Developers: All the team.
- viii. Connect the app to the server. Timestamp: 20 January - 8 February. Developers: All the team.
- ix. Print the result obtained from the server. Timestamp: 20 January - 8 February. Developers: All the team.
- x. Divide the app functionality into fragments. Timestamp: 21 - 25 February. Developer: Ramón Costales.
- xi. Fix navigability. Timestamp: 18 April. Developer: Ramón Costales.

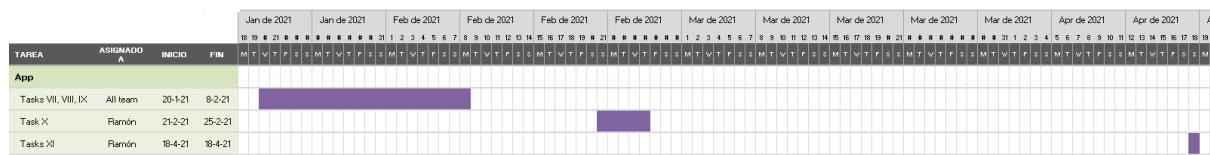
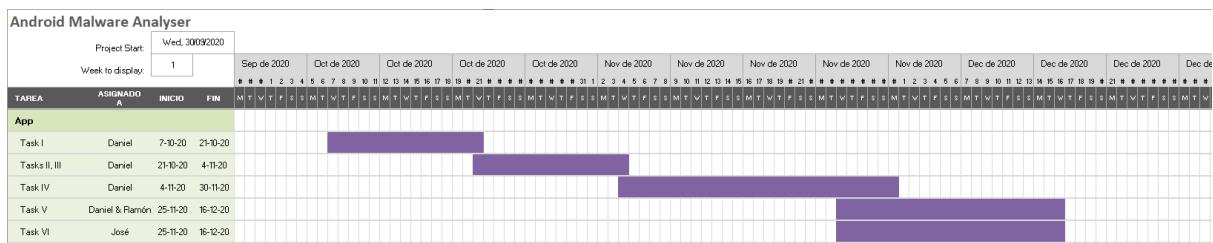


Figure 4: App planning

#### 4. Server (see Figure 5):

- i. Create the server. Timestamp: 20 January - 8 February. Developers: All the team.
- ii. Create a Spark Streaming script that processes the logs. Timestamp: 20 January - 8 February. Developers: All the team.
- iii. Develop the multi connection. Timestamp: 21 March - 2 April. Developer: Ramón Costales.
- iv. Update the Spark Streaming script. Timestamp: 25 - 26 April. Developer: Ramón Costales.
- v. Process, parse and filter the logs. Timestamp: 26 April - 4 May. Developer: Ramón Costales.

- vi. Send back the results to the app. Timestamp: 4 - 8 May. Developer: Ramón Costales.
- vii. Fix bugs. Timestamp: 8 - 11 May. Developer: Ramón Costales.

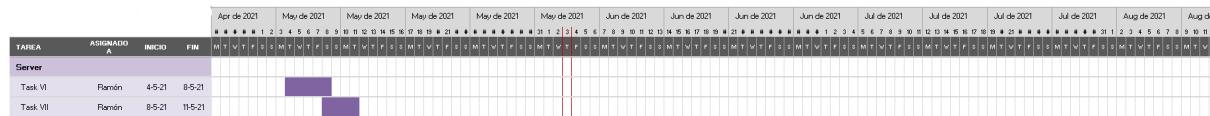
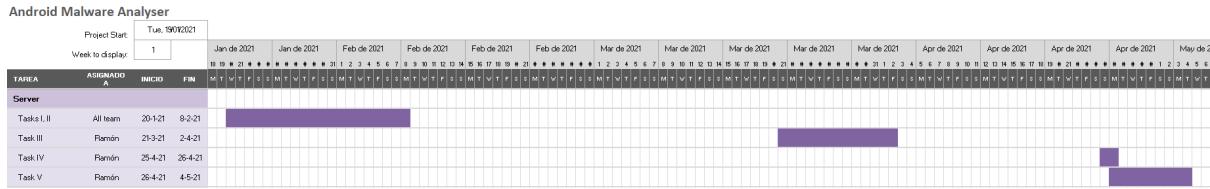


Figure 5: Server planning

## 5. App [server settings fragment] (see Figure 6):

- i. List all the connections. Timestamp: 25 - 26 February. Developer: Ramón Costales.
- ii. Develop the add functionality. Timestamp: 26 - 28 February. Developer: Ramón Costales.
- iii. Develop the delete functionality. Timestamp: 28 February - 2 March. Developer: Ramón Costales.
- iv. Save and load the connections. Timestamp: 2 - 5 March. Developer: Ramón Costales.
- v. Parse the IP and port numbers. Timestamp: 19 - 20 April. Developer: Ramón Costales.



Figure 6: Server settings planning

## 6. App [apps information fragment] (see Figure 7):

- i. Convert the apps list activity to a new fragment. Timestamp: 25 - 28 February. Developer: Daniel Puente.
- ii. Convert the app details activity to a new fragment. Timestamp: 25 - 28 February. Developer: Daniel Puente.

- iii. Develop the permissions view organized by categories. Timestamp: 16 - 23 March. Developer: Daniel Puente.

## Android Malware Analyser

*Figure 7: Apps information planning*

## 7. App [permissions analyzer fragment] (see *Figure 8*):

- i. Permissions research. Timestamp: 1 - 7 March. Developer: Daniel Puente.
  - ii. Elaborate permissions and domains dataset. Timestamp: 8 - 15 March. Developer: Daniel Puente.
  - iii. Design and code permissions grading algorithms. Timestamp: 24 March - 1 April. Developer: Daniel Puente
  - iv. Develop permission analyzer logic and functionality. Timestamp: 5 - 18 April. Developer: Daniel Puente.
  - v. Develop permissions analyzer view. Timestamp: 19 - 30 April. Developer: Daniel Puente.
  - vi. Finish, test, debug and correct errors. Timestamp: 1 - 12 May. Developer: Daniel Puente.

Android Malware Analyser

*Figure 8: Permission analyzer planning*

#### 8. App [signature analyzer fragment] (see *Figure 9*):

- i. Upgrade and upload the first database version. Timestamp: 25 November - 20 February. Developer: José Ignacio Daguerre.
  - ii. Development of the Signature Analyzer fragment. Timestamp: 20 - 27 February. Developer: José Ignacio Daguerre.
  - iii. Creation of an activity with the “Check Hash” option. Timestamp: 28 February- 1 March. Developer: José Ignacio Daguerre.

- iv. Connection and interaction with the database through the application. Timestamp: 1 - 2 March. Developer: José Ignacio Daguerre.
- v. Display all the apps in a checkbox list. Timestamp: 3 - 17 March. Developer: José Ignacio Daguerre.
- vi. Implement a search box in Signature Analyzer Fragment. Timestamp: 1 - 19 April. Developer: José Ignacio Daguerre.
- vii. Finish Signature Analysis fragment. Timestamp: 22 April - 1 May. Developer: José Ignacio Daguerre.
- viii. Upgrade Signature Analysis fragment with an updatable database. Timestamp: 3 - 6 May. Developer: José Ignacio Daguerre.
- ix. Upload the final version of the database. Timestamp: 11 May - 12 May. Developer: José Ignacio Daguerre.

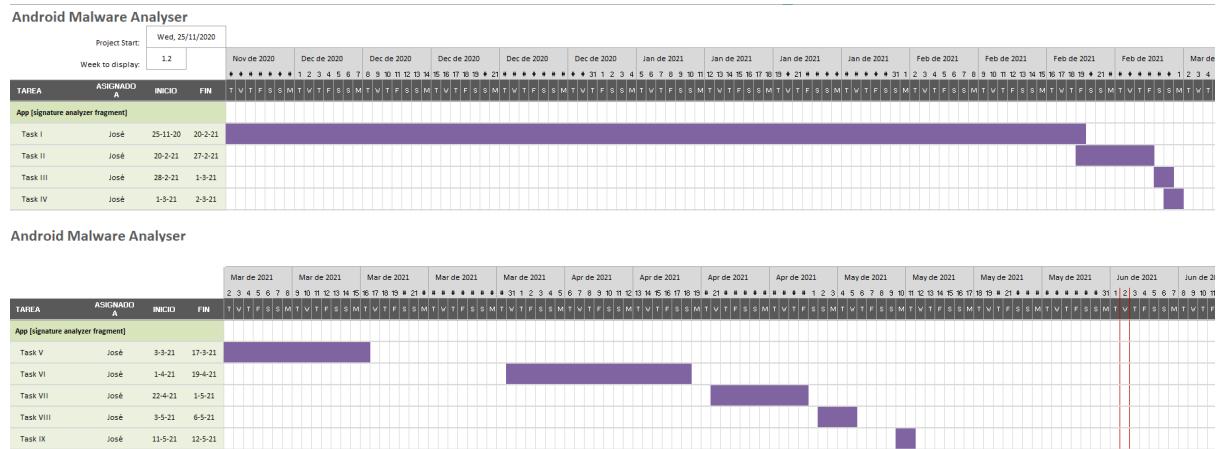
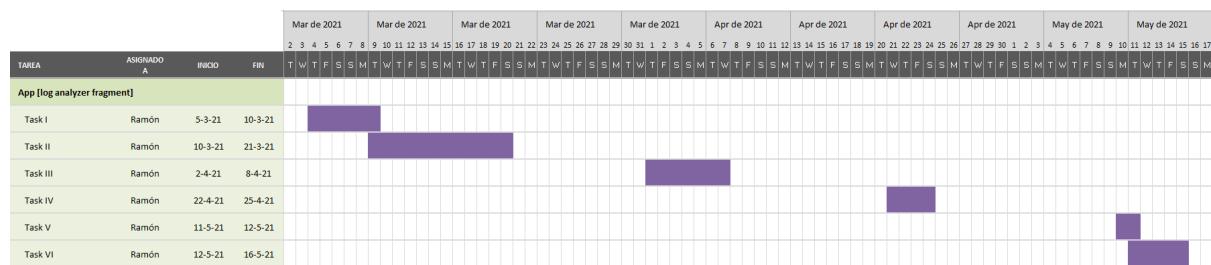


Figure 9: Signature analyzer planning

## 9. App [log analyzer fragment] (see Figure 10):

- i. Display all the apps in a checkbox list. Timestamp: 5 - 10 March. Developer: Ramón Costales.
- ii. Program the connection to the server. Timestamp: 10 - 21 March. Developer: Ramón Costales.
- iii. Clean the code. Timestamp: 2 - 8 April. Developer: Ramón Costales.
- iv. Develop the fragment that shows the log analysis result. Timestamp: 22 - 25 April. Developer: Ramón Costales.
- v. Implement expandable elements while showing the result. Timestamp: 11 - 12 May. Developer: Ramón Costales.
- vi. Check the permissions of the apps when showing the result. Timestamp: 12 - 16 May. Developer: Ramón Costales.

Android Malware Analyser

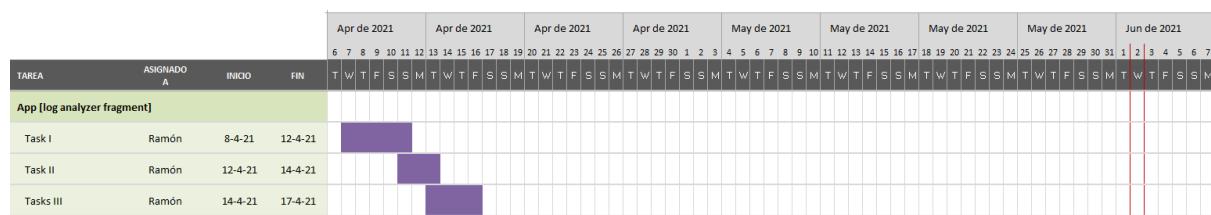


*Figure 10: Log analyzer planning*

#### **10. App [previous results fragment] (see Figure 11):**

- i. Create the PrevResults database. Timestamp: 8 - 12 April. Developer: Ramón Costales.
  - ii. List all the previous results. Timestamp: 12 - 14 April. Developer: Ramón Costales.
  - iii. Develop a fragment that shows the result of an analysis. Timestamp: 14 - 17 April. Developer: Ramón Costales.

Android Malware Analyser

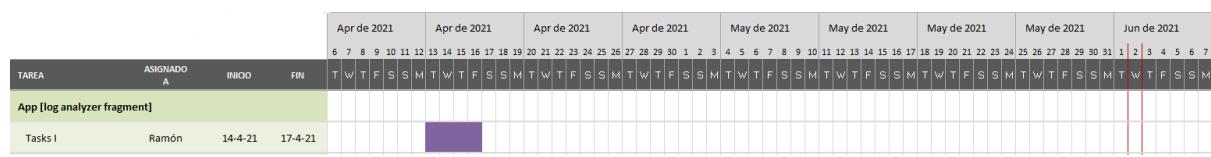


*Figure 11: Previous results planning*

#### **11. App [about us fragment] (see *Figure 12*):**

- i. Create the fragment. Timestamp: 14 - 17 April. Developer: Ramón Costales.

Android Malware Analyser

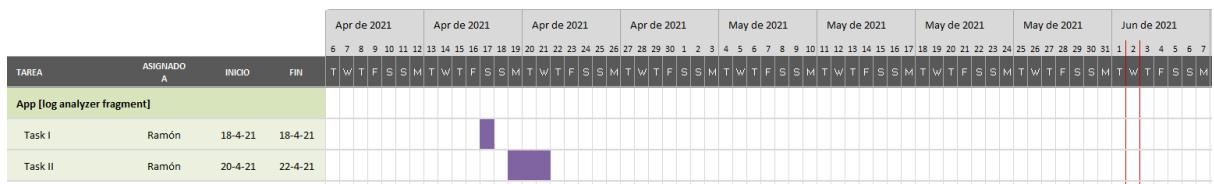


*Figure 12: About us planning*

## 12. App [home fragment] (see *Figure 13*):

- i. Develop the fragment. Timestamp: 18 April. Developer: Ramón Costales.
  - ii. Display elements whose hash is yet to be analyzed. Timestamp: 20 - 22 April. Developer: Ramón Costales.

Android Malware Analyser



*Figure 13: Home planning*

## 1.5 Document Organization

This document is structured in 9 chapters. **Chapter 1** introduces the context of the project. **Chapter 2** describes the Android operating system and reviews its vulnerabilities and the security systems developed for this OS; this chapter ends with a malware overview. **Chapter 3** names all the tools and samples used in the development of this project and argues the decisions we took in each step of its production. **Chapter 4** explains the theory behind the analysis methods we developed for the application and **Chapter 5** shows their implementation. **Chapter 6** covers the user's interaction with the application and its layouts. **Chapter 7** shows the results obtained from analyzing several applications using the application developed throughout this project. **Chapter 8** lists all the individual contributions of each member of the team. Finally, **Chapter 9** reviews the conclusions drawn from this project and brainstorms future work and improvements that we didn't have the time to accomplish.

# 2. Android

This chapter introduces the Android Operating System [1], describes the key characteristics of applications and enumerates all types of components and intents. Also, Android's vulnerabilities are outlined, the rooting method is explained and Android's different security measures are listed. Lastly, an analysis of Android's malware types, characteristics and trends is performed.

## 2.1 Android Operating System

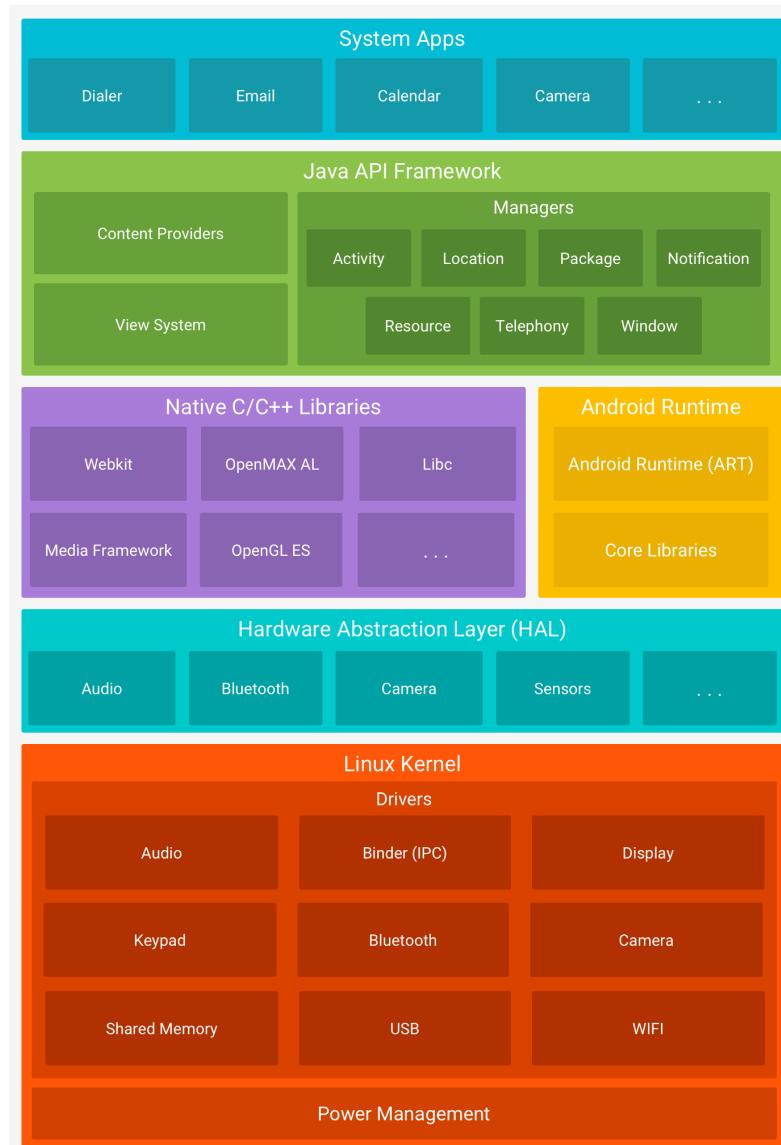
### 2.1.1 Introduction

Android is an Operating System that consists of a stack of open source software based on the Linux Kernel and created specifically for mobile devices. Its architecture is conformed of six layers [21]:

- **The Linux Kernel:** It is the base layer of the Android Operating System, required for carrying out essential functionalities such as process management, memory, network stack, controller model and key security features. It is not the standard kernel, but a specific fork that includes additional elements, such as Binder for inter-process communication, specific drivers, etc.
- **Hardware Abstraction Layer (HAL):** allows the Java API Framework layer to make use of standard interfaces that permit the leverage of device hardware capabilities. It consists of a set of library modules, one for each hardware component, such as the camera.
- **Android Runtime (ART):** Each app runs in its own process and with its own instance of the Android Runtime. It is designed to be able to run multiple virtual machines on low memory devices through DEX files. Before Android 5.0, the Dalvik virtual machine was used. The main difference between Dalvik and ART is that the latter compiles the bytecode files during the installation of the application, so its key objective is to compile the sources in DEX code. It has the Core Library (a particular implementation of the Java API), basic commands accessible through an ADB shell [45], native system daemons and services, and the Init process.
- **Native C/C++ Libraries:** They allow applications to interact at a low level with the kernel. All Android libraries are Open Source. Examples: Bionic (C standard library on Android), WebKit (web page rendering and JavaScript interpreter), SQLite (database), OpenSSL (SSL Sockets), etc.
- **Java API Framework:** Every part of the Operating System is accessible through an API written in Java, which is used for developing applications with the objective of reusing components. It is not exactly a library, since there is an inter-process communication with Binder for limiting accesses for security reasons. Examples: notification manager, packet manager, window manager etc.

- **System Apps:** These are the preinstalled apps on the system that offer functionality for both the user and other apps that may require them.

These layers are detailed in *Figure 14*.



[21] Figure 14: The Android software stack

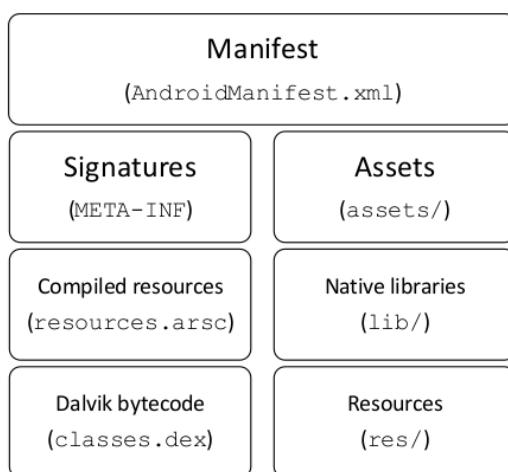
## 2.1.2 Applications

### APK files

Android apps are packaged and distributed in APK (Application Package) files that are based on Java JAR packages. These packages have the following structure [22]:

- **AndroidManifest.xml** – It is the application's configuration file. Several attributes are defined in it, such as the unique identifier of the application, its components («activities», «receivers», «content providers», etc.) or the permissions it requires.
- **classes.dex** – Contains the application's compiled code.
- **resources.arsc** – It is the file containing precompiled resources.
- **META-INF** – It is the directory that stores information corresponding to the digital signature of the application; it contains the following files:
  - **MANIFEST.MF** – It contains a complete list of the APK files along with their respective SHA-1 hash.
  - **CERT.SF** – It contains the SHA-1 hash of every 3 lines that appear in the MANIFEST.MF.
  - **CERT.RSA** – stores the signature of the CERT.SF file and the certificate used to sign the files.
- **res** – It is the directory that stores the resources (images, text files, XML files, etc.) used by the application.
- **lib** – It is the directory that contains the compiled code for different architectures: armeabi, armeabi-v7a, x86 or mips.
- **assets** – It contains non-processed resources.

Figure 15 displays the aforementioned contents of APK files.



[23] Figure 15: APK file structure

As explained in chapters 4 and 5, APK files are of great importance in our project, as we make use of them in all the analysis methods we carry out:

- In the Cryptographic Signature Analysis, we use the entire APK file to compute the hash signature of the installed applications.
- In the Permission Analysis, we read the permissions that applications require from their respective AndroidManifest.xml file.
- In the Heuristic Log Analysis, as in Permission Analysis, we read the permissions that the analyzed applications require.

## Permissions

Of all the files that make up an APK file, the most relevant for our project is the AndroidManifest.xml [24], since it contains the permissions that the application requires. This information is vital for two of the three analysis methods that we perform: the permission analysis and the log analysis.

Android restricts access to specific data and actions in order to protect user privacy. In case an application needs to access any of those restricted components, it has to request the specific permission needed for accessing them. These permissions are categorized in two different ways: categorized by type and by group.

There are several types of permissions, divided by the scope of restricted data or actions that the application may perform once the permission is granted [25]:

- **Install-time permissions:** They give the app limited access to restricted data, and they allow it to perform restricted actions that minimally affect the system and other apps. The system automatically grants the permissions when the user installs the app.
- **Runtime permissions:** Also known as dangerous permissions, they give the app additional access to restricted data, and they allow it to perform restricted actions that more substantially affect the system and other apps. Therefore, they need to be requested before they can access the restricted data or perform restricted actions.
- **Signature permissions:** If the app declares a signature permission that another app has defined, and if the two apps are signed by the same certificate, then the system grants the permission to the first app at install time.
- **Normal permissions:** They allow access to data and actions that extend beyond the app's Sandbox. However, the data and actions present very little risk to the user's privacy, and the operation of other apps.
- **Special permissions:** They correspond to particular app operations. Only the platform and OEMs (Original Equipment Manufacturers) can define special permissions.

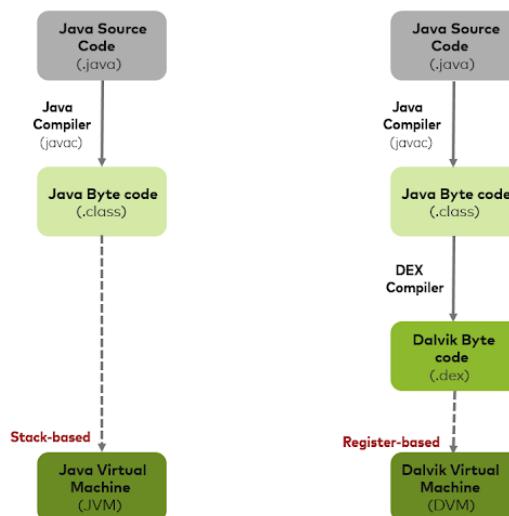
The permissions are divided into groups by their functionality [26]. For example, *android.permission-group.LOCATION* groups the permissions that grant access to the device location, like the permission *android.permission.ACCESS\_FINE\_LOCATION* or the permission *android.permission.ACCESS\_COARSE\_LOCATION*.

The user can control the permissions requested by an application by taking one of the following actions:

- **Allow the permission:** Allows the application to access the specific data or actions requested by the permission all the time.
- **Allow only while in use:** Introduced in Android 10 for location related permissions [27], the application will have access to the device's location only when it is running in the foreground.
- **Allow one time:** Introduced in Android 11 for location, microphone and camera related permissions [28], the application will be granted a temporary one-time permission, which will be revoked after a period of time, after which the user will be prompted for the permission once more.
- **Deny:** The application will not have access to the specific data or actions requested by the permission.

## Execution

On Android, all Java applications run in a separate Linux process, which is also an instance of the Android virtual machine (Dalvik VM before Android 5.0 / ART after Android 5.0). *Figure 16* shows the difference between the execution of normal Java applications and Android Java applications: while Java Virtual Machine receives as input .class files generated in the compilation of Java code, Android's virtual machine receives DEX code as input, which itself is generated from .class files. Regarding applications developed with Kotlin, they only differ from Java applications in the compiler, since instead of using the Java compiler they use the Kotlin compiler, which also generates .class files.

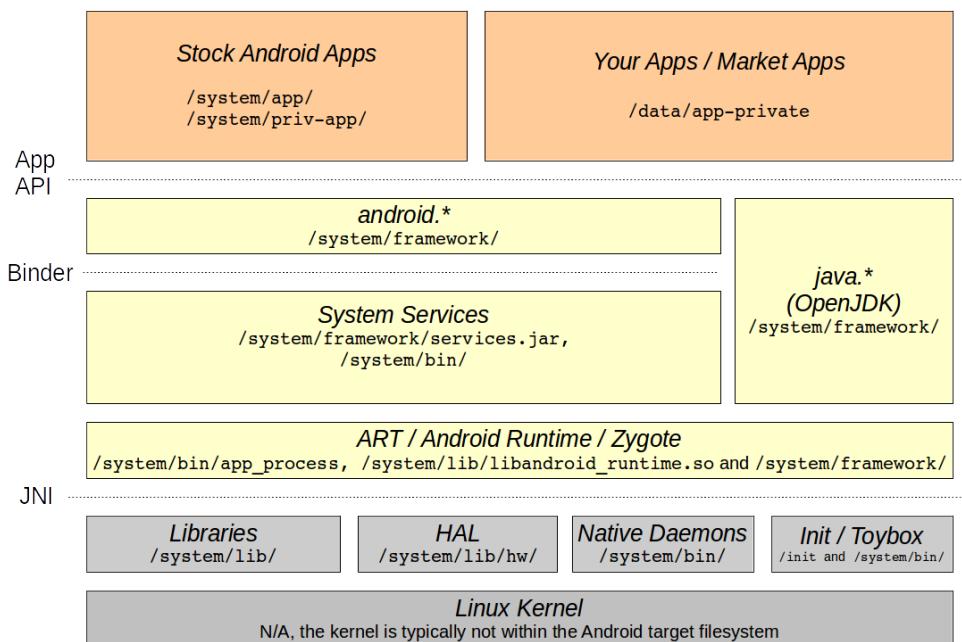


## JVM vs DVM

[29] Figure 16: Java Virtual Machine vs Dalvik Virtual Machine

When an Android application runs, it runs with a process that belongs to a unique UNIX user (it has a unique UID associated with it). This user is created when the application is installed. This implies that each application has its own set of files, configurations and private databases; this is what is known as Sandbox. This form of execution ensures that independence from other apps is guaranteed, improving system security. To allow sharing of this private data between applications, it must be explicitly requested.

*Figure 17* illustrates the communication interfaces between the different Android components, as well as Android's file system:



[30] *Figure 17: Android architecture vs filesystem*

### 2.1.3 Components

Components are the building blocks used to create applications. Each component is a possible entry point to the app. Applications are launched when one of its components is activated through an intent. There are four types of components [31]:

- **Activities:** Each of the screens of the graphical interface. The design is implemented using XML and the response functionality of graphical interface events using Java or Kotlin. An Activity can be either in Running State (fully visible; the user is interacting with it), Paused State (partially visible; for example if a dialog box appears above it) or Stopped State (not visible; the user is working with another application or with another Activity of the same app).

- **Services:** Responsible for launching remote tasks or processes in the background. They allow tasks to continue running while the user performs other actions.
- **Content providers:** They manage a shared set of data. Each provider manages access to a central data repository through a series of permissions, implying that, if any app wants to access its content, it must have the required permissions defined in its manifest file and subsequently be approved by the owner of the device.
- **Broadcast Receivers:** They respond to system-wide message notifications.

## 2.1.4 Intents

Intents are messages that an application sends to the Android system to activate a component of an application. There are two types of intents [32]:

- **Explicit intents:** Activates a specific component of a specific application. For example, clicking an application icon activates its main Activity.
- **Implicit intents:** Activates a system component capable of doing a specific task. For example, clicking on a PDF opens an application capable of reading it.

## 2.2 Android's Vulnerabilities

### 2.2.1 General Vulnerabilities

Users store both personal and business private information on their mobile devices. As a result of this, the need arises to use data protection mechanisms that guarantee the integrity, authenticity and confidentiality of the same.

In Android, applications can be downloaded from unofficial markets. This fact implies that many applications in third-party markets are covert malware. For this reason, one of these protection mechanisms was developed, the so-called Google Bouncer. This service controls the publication of new applications on the Google Play Store to prevent those that are malicious from entering the official market. In its year of release, it was estimated that the number of malicious applications found on the Google Play Store [33] decreased by 40%, achieving a first line of defense. However, lacking high detection rates, large amounts of malware escape its control.

Another defensive measure developed for Android is the Google Play Protect [34]. Leveraging machine learning, every day it scans all the apps on Android devices to avoid the installation of malicious apps, reaching more than 100 billion scanned apps every day. For this reason, it is the most widely deployed mobile threat protection service in the world. It is composed of an on-device protection [35] and a cloud-base protection [36]. On-device protection includes PHA (Potentially Harmful Applications) scanning services, which consists of the aforementioned daily on-device scan (discovers 93% of PHAs), but also an

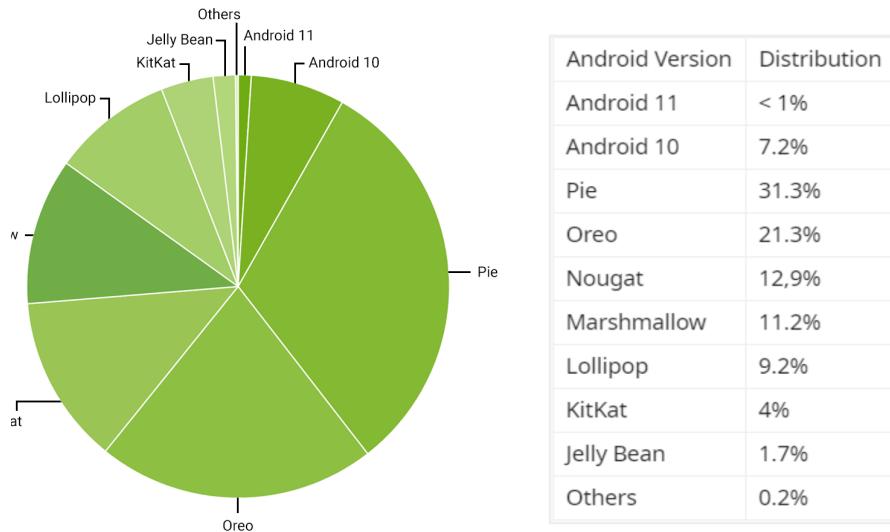
on-demand scan and an offline scan that blocks more than 300 million PHA installations annually. In turn, the cloud-based protection performs static code analysis, signature analysis, dynamic analysis, etc. Despite this set of functionalities, it seems that this service is not as efficient compared to other protection apps on the market. In a 2017 analysis of 18,000 malware apps, Google Play Protect scored the last of 16 security apps, with a 56.8% detection rate [37]. In March 2021 another analysis revealed an increase in its detection rate, since it scored a 70.2% against latest Android malware samples as shown in *Figure 18* [38]. Still, the industry average is a 98% detection rate, so it's still not enough.



[38] *Figure 18: GPP's protection score*

In addition, there is the possibility of self-signing the certificates of the apk files, so they do not require any certification authority to ensure that the application does not pose any risk to the user.

Another problem arises with the establishment of custom permissions. This was approached, but not in its entirety, with the Android 6.x (Marshmallow) release [39], as it moved from an Install-time permissions policy to a Runtime permissions policy. This change meant that the user must accept the permission groups when the application is running. In addition, where previously the user had to accept all permissions to use the application, the user can now reject some of the permissions and still use the application even if it has some functions that are not accessible. With Android 10.x [40] provides an option to accept permissions only when the application is in use, improving users' control over these permissions even further. As can be appreciated, the operating system updates have brought with them a significant improvement in terms of access limitations and restrictions on certain permissions. Despite this, as we can see in *Figure 19*, only 8.2% and less than 1% of systems have upgraded to Android 10 and 11 respectively, and 26.3% of all systems still do not have a version higher than Marshmallow.



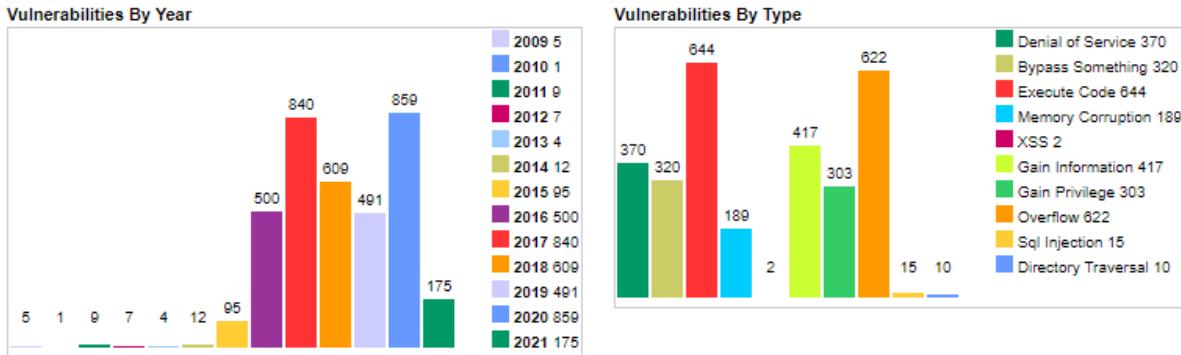
*Figure 19: Android version distribution Feb 2021*

As shown in *Figure 20*, the aforementioned has led to Android topping the list of the most vulnerable operating systems over the last 20 years.

1999-2019		2019	
Debian Linux	3,067	Android	414
Android	2,563	Debian Linux	360
Linux kernel	2,357	Windows Server 2016	357
Mac OS X	2,212	Windows 10	357
Ubuntu	2,007	Windows Server 2019	351
Mozilla Firefox	1,873	Adobe Acrobat Reader DC	342

[41] *Figure 20: Top 20 Products with the most technical vulnerabilities over time*

Fortunately, the number of vulnerabilities is decreasing over the years. As shown in *Figures 21* and *22*, the most relevant vulnerabilities are code execution, buffer overflow, denial of service, information gathering and privilege escalation.



[42] Figure 21: Graphs of Android vulnerability trends over time

Vulnerability Trends Over Time																
Year	# of Vulnerabilities	DoS	Code Execution	Overflow	Memory Corruption	Sql Injection	XSS	Directory Traversal	Http Response Splitting	Bypass something	Gain Information	Gain Privileges	CSRF	File Inclusion	# of exploits	
2009	5	3								1						
2010	1	1	1													
2011	9	1	1		1			1		4	1	3				
2012	7	3	3	2							1					1
2013	4		1	1	1					1	1	1				
2014	12	2	4	1		1		1		1	2	1				1
2015	95	46	49	50	37					13	14	17				
2016	500	104	72	91	38					47	96	236				
2017	840	86	206	170	32			1		30	113	36				
2018	609	32	84	143	12	2	1	2		17	63	3				
2019	491	37	107	41	24	3		1		39	22	1				
2020	859	46	97	104	27	9		5		148	98	3				
2021	175	2	19	19	17			1		19	6	2				
Total	3607	370	644	522	189	15	2	10		320	417	303				2
% Of All		10.3	17.9	17.2	5.2	0.4	0.1	0.3	0.0	8.9	11.6	8.4	0.0	0.0		

[42] Figure 22: Table of Android vulnerability trends over time

## 2.2.2 Rooting

### Introduction

The process known as rooting consists of acquiring temporary root access on the device and then installing a tailor-made “su” binary, achieving persistent root access [43].

There are different methods to temporarily access root, depending on whether the bootloader is unlockable (“pre-root”) or not (“post-root”) [44]. Unlockable bootloader means that it allows you to write custom partition images to the device storage or boot the device from an image not stored on the device. Pre-boot requires the bootloader to boot into a special bootloader mode or a custom OS recovery or from a bootable external SD card. Post-boot cases consist of obtaining a root shell after booting the phone, either through an exploitable privilege escalation vulnerability or a privileged ADB [45].

For devices without an unlockable bootloader, root access can be achieved by exploiting a kernel or system vulnerability. A privilege escalation exploit, typically packaged in one-click rooting applications, allows an application to run a root shell to install the "su" binary or modify system settings.

Another way is via a privileged ADB (Android Debug Bridge) [45]. The system property "ro.secure" of "default.prop" determines the UID (User ID) of the process under which an ADB shell is executed. When the value is 1, the daemon process "adbd", which initially runs as root, changes its UID before creating the ADB shell without root privileges. Otherwise, users can have a shell that can run any program as root.

## Root Vulnerability

Android's permission system forces access controls on security-related resources such as sensors, sensitive data and important communication modules. But if the phone is rooted, this permission system can be avoided. On a rooted phone, processes can run with root privilege and it is possible to access any resource without permission. Many people root the device to uninstall stock apps, flash third-party ROMs, use applications that require root permission, back up the phone...

The problem is that when dealing with a rooted mobile phone, malware can access sensitive databases (SMS, Contacts...) and hardware interfaces (camera, microphone...) without having the corresponding permissions beforehand. In these cases, the permission system is not relevant, because it is avoided.

Whereas there are applications that offer one-click-root (by clicking a button they are able to root the phone), equally there exist applications that offer one-click-unroot (it removes the "su" binary). Removing root from the phone takes away root permissions from potential malware, so the permission system becomes relevant again, denying malware access to system resources. However, during the time window in which the device is rooted, if the malware has modified the packages.xml file (containing a list of permissions and packages) or apks with root privileges, it may have escalated its permissions, causing permission escalation after root removal to be a backdoor for the malware to abuse resources [46].

They could also delete the certificate restriction for sharing UIDs with another app; if this is done with a privileged app, then privileged permissions of that app or access to its data can be obtained. Moreover, the code could also have been modified to remove the permission access control.

Therefore, this type of malware offers a higher level of impact, as they are able to persist after root removal, as well as having a very high detection evasion rate.

## 2.3 Android's Security Systems

### 2.3.1 Linux Security

At the operating system level, the Android platform uses Linux kernel security such as secure inter-process communication (IPC) to enable secure communications between applications running in different processes [47]. This ensures that even native code is restricted by the application Sandbox. Thus, the system is designed to prevent a malicious application from damaging other apps, the Android system, or the device.

The Linux kernel provides Android with several key secure features, including [47]:

- A user-based permissions model.
- Process isolation.
- Extensible mechanism for secure IPC.
- The capability to remove unnecessary and potentially insecure parts of the kernel.

A fundamental goal of kernel security is to isolate the resources of one user from those of another user, thus [47]:

- Prevents one user from reading another user's files.
- Ensures that one user does not exhaust the memory of another.
- Ensures that one user does not drain another user's CPU resources.
- Ensures that one user does not drain another user's devices (telephony, GPS, Bluetooth...).

### 2.3.2 Application Sandbox

The security of Android applications is enforced by the application Sandbox [84], which isolates applications from each other and protects apps and the system from malicious apps. It achieves this by assigning a unique user ID to each app and running it in its own process.

The kernel enforces security between apps and the system at the process level through standard Linux facilities such as user and group IDs that are assigned to different apps. By default, apps cannot interact with each other and have limited access to the OS. As the application Sandbox is located in the kernel, this security model extends to both system applications and native code. All software above the kernel, such as operating system libraries, application frameworks, application runtime (ART), and all applications, are running inside an application Sandbox.

Generally, to evade the application Sandbox on a properly configured device, kernel security must be compromised. Nevertheless, the individual protections that force the application Sandbox are not invulnerable, so protection into depth is important to prevent a single vulnerability from compromising the operating system or other apps. With each Android version, protections have been added to protect the application Sandbox, such as in 9.0,

which forced all non-privileged apps to run in individual SELinux Sandboxes, providing mandatory per-app access control, to improve the separation of apps, prevent overwriting of secure defaults, and prevent apps from making their data accessible to everyone.

It is not a good idea to make data accessible to everyone, as this can be an information leak and a popular target for malware. From Android version 9 onwards this is not allowed. Therefore, for file sharing it is used by content providers or MediaStore class for those media files that should be accessible to everyone.

### 2.3.3 SELinux

Android uses Security-Enhanced Linux (kernel security module) [48] to apply access control policies and set mandatory access controls on processes. It applies Mandatory Access Control (MAC) instead of Discrete Access Control (DAC). This implies that instead of the owner of a resource controlling the access permissions attached to that resource, any access is queried to a central authority. This ensures that the software runs only at the lowest privilege level, mitigating the effects of potential attacks.

### 2.3.4 System's Security

The system partition contains the Android kernel, as well as system libraries, the application runtime (ART), the application framework and applications [47]. The partition is read-only. When the device is booted in safe mode, third-party applications can be launched manually by the device owner, but are not launched by default.

File system permissions ensure that a user cannot alter or read another user's files, unless the developer explicitly shares files with other applications [47]. In Android, each application runs as its own user.

Verified boot ensures the integrity of the device's software, starting from a hardware root of trust till the system partition [87]. During boot, each stage cryptographically verifies the integrity and authenticity of the next stage before executing it. This makes privilege escalation non-persistent, because it detects file system modifications and compromised devices are not allowed to boot.

Android provides a set of cryptographic APIs for use by applications [47]. This includes implementations of standard and commonly used cryptographic primitives, such as AES, RSA, DSA and SHA. Additionally, these APIs can be used by high-level protocols, such as SSH and HTTPS. It also has a KeyChain class that allows applications to use system credential storage for private keys and certificate chains.

By default, only the kernel and a small subset of core applications can be run with root permissions. Android does not prevent a user or application with root permissions from modifying the operating system, kernel, or any other application. In general, root has full access to all applications and their data. Users who change permissions on an Android device to allow root access to applications increase their exposure to malicious applications and potential application crashes.

### 2.3.5 User's Security

Android supports full file system encryption, so all user data can be encrypted in the kernel [47]. It also allows full disk encryption, so that a single key (protected by the device password) protects the entire user data partition; at boot time the user must provide credentials before any part of the disk becomes accessible. It also supports file-based encryption, allowing different files to be encrypted with different keys that can be unlocked independently.

Encrypting data with a key stored in the device does not protect application data from users with root permissions. Applications can add a layer of data protection by using encryption with a key stored outside the device, such as on a server, or a user password. This provides temporary protection while the key is not present, but at some point the key must be given to the application, making it accessible to users with root permissions. A more robust approach to protect data from possible access by users with root permissions is the use of hardware solutions. Manufacturers can implement hardware solutions that limit access to specific content.

Android also allows for pre-access verification of the device through a password given by the owner. Not only does it prevent access, but it also protects the cryptographic keys for file system encryption.

In the case the device is lost or stolen, the encryption of the entire file system uses the device's password to protect the encryption key, so that modifying the bootloader or operating system is not enough to gain access to the user's data.

### 2.3.6 Apps Security

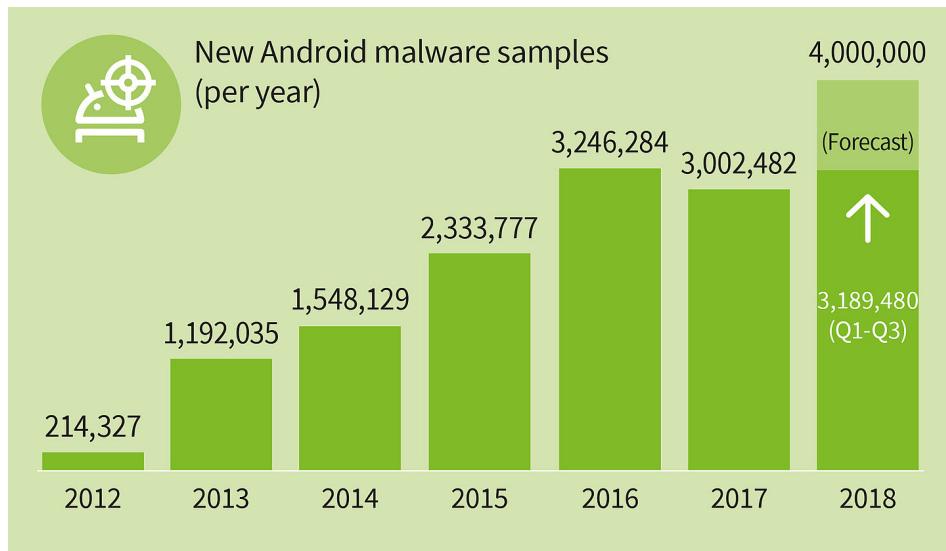
Applications can only access a limited set of resources managed by the OS [49]. Nevertheless, applications usually need access to a different set of resources outside the Sandbox such as the camera or Bluetooth. This requires the use of protected APIs, which are intended to be used by applications through permissions. Since permissions are managed by the OS, in order to use these APIs, applications must define the permissions in the manifest and then the device owner either accepts or rejects them during installation. In case an application tries to access a protected API not declared in the manifest, a security exception is raised and returned to the application, denying its access to the requested resource.

## 2.4 Android's Malware

### 2.4.1 Malware trends

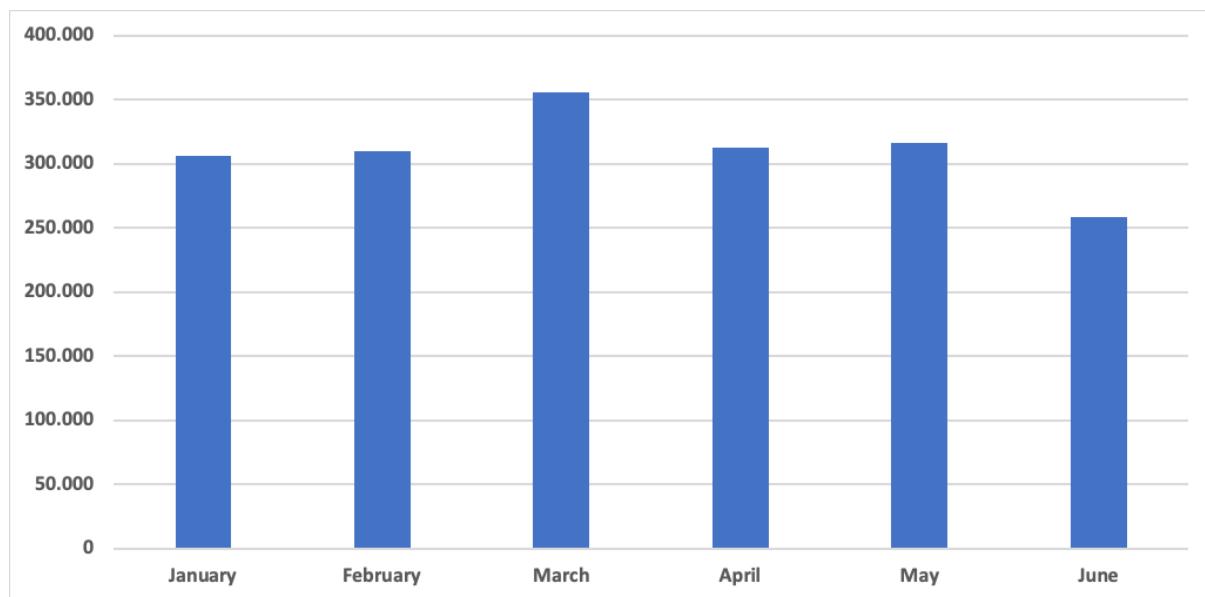
Malware development for mobile devices has increased considerably in the last few years. In 2019, there were already more than 27 million malware programs in the Android mobile

sector [50]. A growth of 690,000 new malware programs was observed, resulting in an increasing number of botnets targeting Android systems [50]. Most infections are due to malicious apps obtained from third parties, which has increased by around 85% per year since 2011 [50]. *Figure 23* below shows the growth of malware samples on Android from 2012 to 2018.



[40] *Figure 23: New Android malware samples per year*

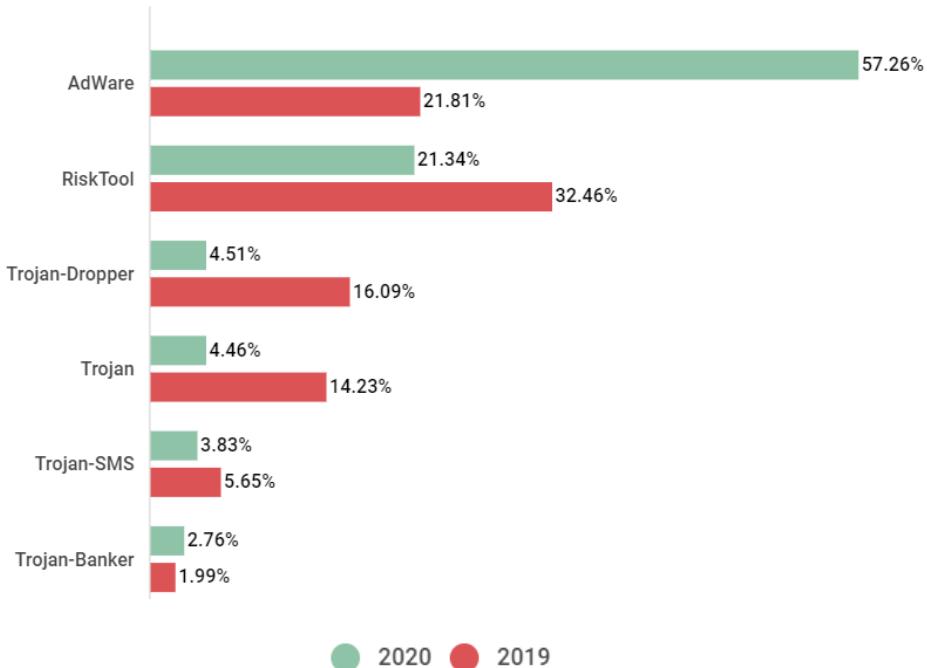
In 2020, threats on Android devices are divided into four different categories [51]: Malware, which accounts for approximately three quarters of the total; Adware, which represents 15.4% of the total; Riskware and PUA (Potentially Unwanted Applications) being almost negligible at 6% and 4% respectively. As shown in *Figure 24*, in 2020 malicious activity increased by 30% in March, which coincided with COVID-19 crisis [52]. As workplace work has been forced to move to home, much of the workload has shifted to home, which is often less protected than a company's network.



[53] Figure 24: The Android threat activity in Q2 compared to Q1 2020

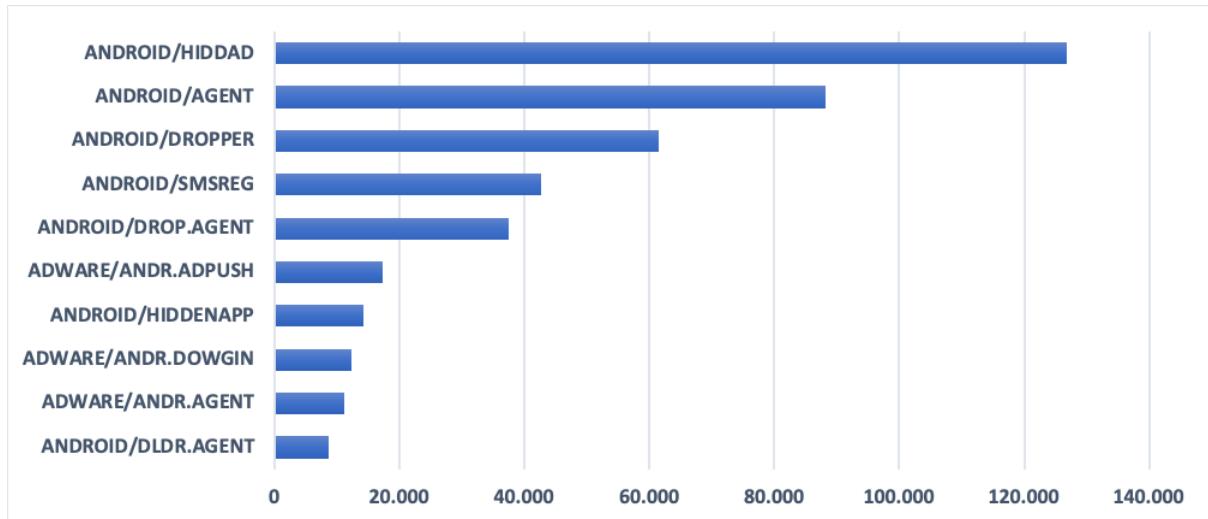
Figure 25 illustrates the top mobile threats detected by Kaspersky in both 2019 and 2020. It shows that the use of Adware has doubled in a single year.

#### Types of mobile threats



[54] Figure 25: Distribution of new mobile threats by type in 2019 and 2020, Kaspersky

Out of the top ten malware families detected, five of them make intrusive use of ads, with Android/Hiddad topping the list [53]. It is worth mentioning that Trojan droppers represent a third of the top ten families detected in the second quarter of 2020, as shown in *Figure 26*.



[53] Figure 26: Top ten detected families in Q2 2020

## 2.4.2 Malware types, aims and characteristics

Common mechanisms:

- **Persistence:** Usually, malware samples seek to persist on the device. One of the most common mechanisms to achieve persistence is through component hiding. One possible technique to achieve this objective is to disable the activity component registered in the Launcher by the application at the moment of its installation, as a result of which the application's icon disappears. To further enable code execution, malware should implement a receiver component to log system events and a service component for background execution. This allows the malware to run in background at system events, such as start-up or WiFi activation, even though the icon is not visible. Examples of this can be seen in samples of spyware, RATs, clickers or ransomware, as they can deploy their full functionality from background execution.
- **Denial of service:** Malware seeks to block access to screens from which the application could be removed. It does this by using services that, while running in the background, detect when the application is trying to be uninstalled and overlay a component that prevents it from doing so. To achieve this, they usually implement the *GET\_TASKS* permission to get the application that is running in the foreground, as well as the *BIND\_DEVICE\_ADMIN* permission to register as Device Administrator.

This section lists the main categories of malware that can be found on the Android operating system. Each category describes its objective and strategies for identification by the analyst [55].

## Adware

This is the most common type of malware on Android devices. When faced with applications with ads, it is controversial to classify them as malware, as it is difficult to establish a limit at which the use of ads starts to be abusive or simply another monetization system.

The main feature of this malware is the inclusion of API keys in the `AndroidManifest.xml` file to obtain the functionality of various ads services, such as AdMob, Baidu, Adwhirl or Ad-X. These API keys contain the identifiers that the ads services use to identify which app is displaying the ads, thus enabling monetary reward.

Another characteristic used by more aggressive samples comes from the inclusion of permissions such as:

- `SYSTEM_ALERT_WINDOW`: It overlays the current window with another one of your choice.
- `GET_TASKS`: Allows you to see what other application is running. If adware detects that a browser is being used, it can redirect the user to an ad page.

## Phishing

The aim of this type of malware is to steal sensitive user information (usually username and password) by deception, pretending to be a legitimate application that hides malware.

Detecting phishing is relatively easy if it tries to pass itself off as a legitimate application. In such a case, by comparing the digital certificates, we can check whether they are applications programmed by the same developer or whether we are dealing with a case of impersonation, causing it to fall into the category of phishing.

Malware of this type requires permissions that allow it to send stolen information, such as access to the Internet, SMS, etc. This allows us to identify this malware, especially if it should not require these permissions given the functionality it promises (a social networking application should not need access to SMS). Also, as they are copies of other applications, it is common for there to be discrepancies with the original or functional errors.

## Spyware

This category covers all types of applications which seek to steal information from a device, such as phone number, email account, contacts, location, installed applications, calls, messages, microphone access, device ID, operating system, MAC address, etc.

In its code, if there is no obfuscation, there probably are strings related to information they are looking for, such as email, location, model, phone, SMS, etc. During execution, data is frequently either sent to a server, posted on a forum or sent by SMS, which involves network traffic. Data may be sent plain or encrypted, making it difficult to identify.

The permissions required by the spyware depend on the information that needs to be extracted, for example:

- *ACCESS\_WIFI\_STATE*: It searches for network information from the device.
- *READ\_CONTACTS*: Access contacts.
- *ACCESS\_COARSE\_LOCATION* o *ACCESS\_FINE\_LOCATION*: Para acceder a la localización.
- *READ\_SMS* o el *RECEIVE\_SMS*: Access messages.
- *PROCESS\_OUTGOING\_CALLS* y el *READ\_PHONE\_STATE*: Phone calls.

## RAT

RAT stands for Remote Access Tool or, if it is hidden inside another application, Remote Access Trojan. The aim of this type of malware is to gain remote control of a device. These actions can be: accessing web pages, installing applications, sending SMS, sending user information, changing device configurations, etc.

As control is remote, the application must communicate with a C&C (Command & Control) server. Commonly, these servers give the malware developer the opportunity to distribute commands to specific devices. It is for this reason that the ultimate goal of this type of malware is the creation of botnets that allow them to launch distributed attacks or black hat SEO (Search Engine Optimization) techniques (they are used to improve the positioning of a website in the search engine results list). This malware normally requires as many permissions as possible, allowing for a wider range of actions.

## Keyloggers

This type of malware collects keystrokes that have been pressed by the user and sends them to an external server. Some controversy also arises with applications with this functionality, as there are keyboard applications that collect keystrokes and statistics to improve their services, raising a dilemma as to whether they should be considered malware or not.

Such applications usually have the following permissions:

- *BIND\_INPUT\_METHOD*: Must be required by an InputMethodService, to ensure that only the system can bind to it.
- *ACCESS\_NETWORK\_STATE*: Allows applications to access information about networks.
- *INTERNET*

## Tapjacking

Malware of this type is designed to trick the user into pressing on the screen, performing a different function than the one the user thinks she/he is performing. The two most typical implementation techniques are based on *Toast* and *WindowManager*.

*Toast* is a system for displaying text messages in pop-up format. Clicks on it are non-functional, so they affect whatever is underneath the pop-up. These messages can be designed using XML, so they can be made to look similar to a dialogue with buttons. This would allow the malware developer to design a pop-up which guides the user's taps to where she/he wants them to go. It is common for such applications to require the *GET\_TASKS* permission to know which application is open and thus which application the user's taps on the *Toast* are working on.

## Clickers

The purpose of this kind of malware is to load web pages and click on links to improve the ranking of that page (black hat SEO), in ads with the aim of creating a large number of hits that generate a financial benefit for the developer, or redirect a victim to download other malware.

It is common for many of the click-accounting systems on websites to be JavaScript code. Clickers must therefore have the ability to load HTML code and interpret JavaScript. Some declare the *SYSTEM\_ALERT\_WINDOW* permission.

## Ransomware

Its aim is to inhibit access to device resources, typically to demand a financial payment. On computers it is usually implemented by encrypting files; Nonetheless, on Android this method is less common due to the application Sandbox, which limits the resources that each application can access, even if a device is rooted , the ransomware could escalate privileges and gain access to all files.

Therefore, the most common in Android are activity blockers, which require permissions to identify the application that is in the foreground and overlap with it, causing the user to be unable to use their device.

These applications usually require the following permissions:

- *RECEIVE\_BOOT\_COMPLETED*: Launch ransomware as soon as the device boots up.
- *USER\_PRESENT o SCREEN\_ON*: Detect if the user is interacting with the device.
- *WRITE\_SETTINGS*: Modify settings on the device.
- *BIND\_DEVICE\_ADMIN*: Allows SystemUI to request third party controls.

## Premium Services

This malware is developed for financial benefit by making the user consume paid services without being aware of it. The ways to achieve it may vary, from forcing the user to subscribe to premium SMS services, to making unwanted calls to premium rate numbers. During execution, unwanted outgoing calls and SMS appear, and it is also common for the user to stop receiving SMS messages because they are being handled by the malware.

In the first case, we would encounter these permissions:

- *READ\_PHONE\_STATE*: Allows read only access to phone state.
- *READ\_SMS*: Receive a code by SMS.
- *WRITE\_SMS*: Send an SMS message with code.
- *INTERNET*

In the second case, these are the permissions required:

- *CALL\_PHONE*: To be able to make phone calls.
- *PROCESS\_OUTGOING\_CALLS*: Identify a number currently being called.
- *WRITE\_SETTINGS*: Changing the system configuration.

### 2.4.3 Transmission methods, detection and prevention

#### Transmission methods

The ways in which Android can be infected are very diverse thanks to all the communication technologies that mobile devices usually have. Below we have listed several ways in which malware can reach the system:

- Downloading infected applications: From the Google Play Store, 3rd party markets such as Aptoide or Internet.
- SMS/MMS
- Bluetooth
- USB
- NFC
- Barcodes
- QR
- Unsecured wireless connections
- Erroneous GSM/UMTS/LTE radio package handling
- Webs
- PC-Phone connection
- Emails

## Detection

There are several ways in which malware can be detected, among them:

- **Excessive battery consumption:** This happens because the malware is resource-intensive.
- **System is slower than usual:** It occurs because the malware is resource-intensive.
- **Excessive data consumption:** This occurs because malware may be constantly communicating (sending or receiving information) with an external party.
- **Lots of advertisements or pop-ups:** Many malware send advertisements to infected users so that the developers can earn revenue through clicks on the ads.
- **Hash, signatures and correlation with other malware:** If we have a database with samples or signatures of malware, by checking an application against this database we can know if it is malicious.
- **File/PackageName:** It is common for new malware to be based on an old one, so they may share file or package names. Another way of detection is for the type and number of system calls; the malware's behaviour is often different from benign applications, so by monitoring logs or system calls and contrasting them, we can get an idea of its intentions. They usually make many calls at the beginning of execution.
- **Malicious dropper:** If we find an APK inside another one, it may be an apparently benign application that contains a malicious installer or dropper.
- **Network addresses assigned in the source code:** Used for command and control attacks.
- **Payload on update:** If we detect unusual changes when updating an application, it may be exploited to download malware.
- **Encrypted files:** Several malware samples try to hide by saving the encrypted malicious payload in the resource folder.
- **Unusual permissions:** In case the application requires permissions which it should not require or does not require permissions which it is supposed to require, it is possible that it is malware, as its functionality is not as expected.
- **Downloads and ratings:** In case the user is trying to download a known app but the app marketplace observes that it has fewer downloads or ratings than it should, it may be a repackaged app masquerading as a malicious version.
- **Installed apps you don't recognize:** They may have been installed without your consent.
- **Redirects to pages without the user's intention:** For example to force the user to view ads or download more malware.
- **Higher-than-normal phone charges:** Because some of them make the user subscribe to premium SMS services without their consent.

Dividing these techniques, we obtain various forms of analysis:

- **Static analysis:** This consists of studying the APK file without the application being executed: AndroidManifest file [24] containing the permissions, Java source code and necessary resources, such as databases or images.

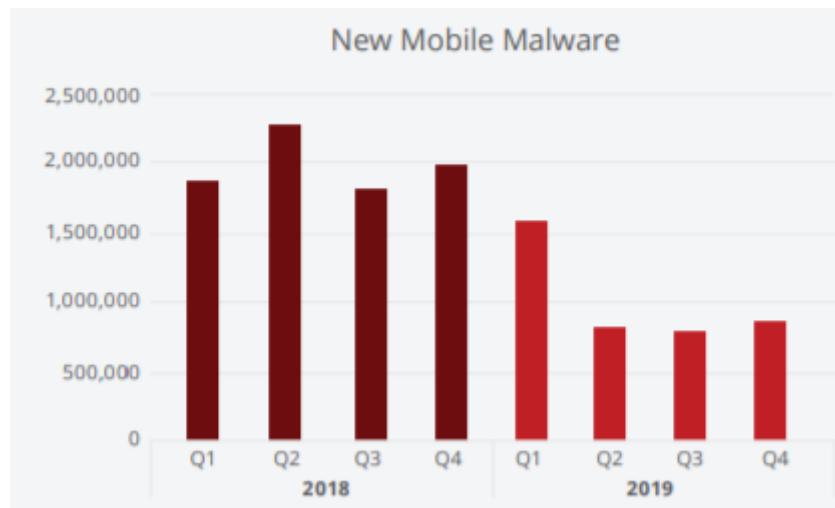
- **Dynamic analysis:** It consists of monitoring the system behaviour to extract the activity of installed and running apps through system calls, user interaction, changes in the file system, use of hardware components or network traffic. From this knowledge, the detector can be trained to learn to distinguish legitimate from malicious behavior.
- **Mixed analysis:** It is based on the combination of the two previous analyses.
- **Metadata analysis:** Consists on analyzing the app's description, its rating in the download market, identification of the creator, to which category it belongs, permissions requested, package name, promotional video, contact website, price, last time it was modified, etc.

## Prevention

Some of the existing ways to prevent malware from entering into a device are:

- **Download apps only from official marketplaces:** Since most malware is found in third-party marketplaces which do not have any kind of filter to avoid publishing it.
- **Connect only to trusted networks.**
- **Download and use of an antivirus.**
- **Disable the downloading of applications from unknown sources option.**
- **Monitor permissions:** Some applications require permissions which are not relevant to the functionality of the application, e.g. to obtain user information.
- **Update OS and applications:** As unresolved vulnerabilities can be exploited in software with outdated versions.
- **Check emails for truthfulness:** There are emails which are often the main transmission method for phishing attacks.

A possible solution to avoid the risk of infection is to make use of the antivirus available in the official Android market (Google Play Store). Unfortunately, several studies have shown that such an approach is not sufficiently effective, as in most cases detection rates are far from optimal [56][57]. However, its use should not be completely ruled out, as there have been some recent tests showing improved detection in certain antivirus engines [58][59]. Moreover, as we can see in *Figure 27*, the detection of recently developed malware on mobile has been decreasing, proportional to the increased complexity of the malware.



[60] Figure 27: New mobile malware detections by quarter 2020

So, for the moment, the effectiveness of antivirus software is questionable, which is one of the reasons why more than half of us do not have an antivirus installed in our Android device. In fact, the total number of downloads of all the antiviruses present in the Google Play Store reaches approximately 1,252,890,800. Note that this figure does not even reach half the number of Android users, taking into account that a single person may have downloaded more than one antivirus, so the number of people with an antivirus installed is even lower.

This antivirus problem is further compounded by a lack of education on mobile security issues. Although computer users are generally aware of the risks inherent in inappropriate or careless use of their device, they do not seem to extrapolate this knowledge properly to the mobile device.

# 3. Workspace

This chapter enumerates and describes the tools used for developing the application, as well as the databases from which the data samples needed for the different analyses have been obtained. This chapter also presents the decisions made throughout the development of the app.

## 3.1 Tools and samples

### 3.1.1 Tools

#### **Android version**

The Android operating system is constantly being updated. In each release, new features can be developed, security measures can be changed, etc. At this moment, the newest release is Android 12 Beta.

We decided to use Android 11 to develop our app as it was the latest release at the time, which ensured that our app was up to date with Android standards.

#### **Android Studio**

Android Studio is the official integrated development environment (IDE) for Google's Android operating system, built on JetBrains' IntelliJ IDEA software and designed specifically for Android development [61].

This tool was used in this project to develop the entire application. It gives the developer the choice to code in Java or in Kotlin. Because of the previous knowledge of Java that the members of the team possess, this app was written in Java.

This IDE incorporates its own Android virtual machine to test and debug the application. However, in this project we decided to test it on a real Android device because of the efficiency and the faster execution time compared to the virtual machine. The IDE also has a GUI editor that helps the developer to edit the .xml view layouts.

#### **EC2 Instance**

We made use of cloud service AWS for the creation of an EC2 instance [62] with at least 4 vCPUs (m4.xlarge) to be able to evaluate parallel implementation within each node with the aim of reducing the cost of monitoring logs locally. It offers scalability, as with a cloud platform you can have simple solutions and then move to more complex ones as your application requirements increase.

## **Spark Streaming**

Spark Streaming is used to process real-time data in a scalable way, with high performance and fault tolerance.

This API is used in the analyzer.py script, developed as part of this work. This script is executed in the server to save the logs received, reduce them and save the logs reduced.

## **SQLite Database**

SQLite [63] is an open source SQL database which stores data in a text file on a device. Android Studio, it comes with a local SQLite database implementation already built in. We decided to use a local database because querying the database in Cloud can take more time, as there can be *time-out* problems in the responses as well as unavailability of the application's service. Although we have decided to store our database locally, in case we need to store large amounts of malware signature datasets and we need more space, we proceed to migrate our database to Cloud.

## **ADB**

Android Debug Bridge (ADB) [45], is a command line tool that allows the user to communicate with the Android device through a console from the PC.

For this project, it has been necessary to do an analysis of all the logs that Android launches, so we must prepare the device to take advantage of all its functionalities. For example, to access all the logcat logs, the android.permission.READ\_LOGS permission must be granted to the analyzer app. This permit is of the special category due to its potential danger, which is why the permission to read logs should only be granted to the applications that are part of the system firmware. To get our application to benefit from this permission we must use the ADB tool.

The ADB command allows you to perform a variety of actions on the device, such as installing and debugging applications. In addition, it provides access to a Unix shell that can be used to run various commands on the device.

For example, the command “adb shell pm grant com.example.androidmalwareanalyzer android.permission.READ\_LOGS” used in the ADB console, allows to grant permissions to the malware analyzer. It is worth mentioning that to use this tool it is necessary to have the USB Debugging of the device activated.

## **Android Virtual Machine**

An Android emulator was required in order to test the analyzer with real malware. The machine has been configured with the virtualization environment VMware, and the version of the operating system is Android 8.

## **PackageManager and PackageInfo**

In order to handle the metadata information that the installed applications contains, the analyzer uses the classes *PackageManager* and *PackageInfo*.

The *PackageManager* is a class for retrieving various kinds of information related to the application packages that are currently installed on the device [78].

The *PackageInfo* is a class that contains overall information about the contents of a package. This corresponds to all the information collected from *AndroidManifest.xml* [77].

## **Github Desktop**

Github Desktop is an application that enables users to interact with GitHub using a GUI instead of the command line or a web browser [64].

We used this tool to manage the versions of the application and to seamlessly merge each contribution of all the participants of the team, allowing a parallel development.

### **3.1.1 Samples**

In addition to the tools necessary to develop the work presented here, we have had to look for samples of malicious applications or permissions misinformation to test the tool.

## **Malicious Apps Hashes**

Despite efforts to acquire a database containing the digital signatures (hash function) encrypted in different encryption algorithms (SHA, MD5, etc.) of all existing malware, we could only find a list of MD5 hashes of malware samples [76].

Nowadays most common encryption algorithms for digital signatures of applications are SHA1, SHA2, and MD5. Therefore, we tried to obtain a dataset which contains one of these algorithms. It is worth mentioning that the most secure option is the use of SHA256 algorithm or higher as it causes less collisions, so we have therefore adapted the Cryptographic Signature Analysis so that it remains functional when using a different hashing algorithm.

## Permissions Dataset

The permissions dataset is mostly used by the permission analysis but it is also used in other aspects of the application. This sample has been made from various sources, specifically from the permissions API reference page of the Android Developers Official Site [79] and from the Android Permissions site [80] so as to get a dataset as complete as possible. What is stored in this dataset are the permission constants, the level of danger, a description of the permissions and the group they belong to.

## Domains Dataset

The Domains dataset is also used by the permission analysis. This sample has been made from the Android Permissions site [42] where all the permissions are assigned to a specific group. In this dataset it is stored the permission groups, an alias of the domain and a description of the group.

## 3.2 Decisions

In this section we present a summary of the design decisions that we have made throughout the development process of this work and that affect the final form of the developed tool.

- We decided to implement a **log analysis for Android** because we wanted to have a dynamic analysis to add some functionality over the two other methods of analysis. Also, since we did not find much information about it, we wanted to try and create something relatively new.
- We opt to **implement a permission analyzer** because it can give the user plenty of information about what the application is trying to achieve. Also, by giving a score we believe we can show visually and effectively if it is actually a benevolent application. Finally, we think that it can raise the user's awareness regarding permission granting and encourage them to check the permissions of the applications being installed on their device.
- We chose to **implement a signature analyzer** since if there is a collision found with its signature, it is almost certain that it is malware, implying that this analysis method gives an almost absolute certainty to the user.
- We decided to **implement a Server Settings Fragment** because every time the EC2 instance is launched, a new IP is set for the instance. If we had not developed this fragment, each time we launched the instance we would have needed to write the new IP in the application code, build the APK and install it on our mobile phones. Also, this fragment allows the user to create their own server and only have to worry about adding the IP without changing any code.
- We chose to **process the logs on the cloud** because doing it locally would have implied that the application would probably lag or crash due to the amount of processing power needed. It also implies that the battery usage is reduced, since less power is needed. Finally, this allows the user to set up their own server for processing the logs.

- We also concluded that we would **only analyze the applications installed by the user** and not the stock applications since these applications are developed by relevant companies which are globally trusted. Also, since the user cannot uninstall them, it would only mean that the analyses would take more time and they would not gain anything.
- We decided to **implement the upgrading and adjustment of the application code in case of a database replacement**, because in case of being able to acquire a data set with encryption algorithms better than MD5 or even containing several types of algorithms applied for a single application.
- We decided to **develop our app for at least Android 8.x (Oreo) versions**, as they moved from an Install-time permissions policy to a Runtime permissions policy. In addition, we make sure we are up to date with version 11, which can be considered the most up to date version, as version 12 is still in testing. Finally, we decided to work with this version because, in the permission analysis, the app retrieves the category of the installed apps. This action can only be performed with a 26 API level which corresponds to Android 8.

# 4. Analysis Methods

This chapter introduces the theory required for understanding the three analysis methods we have performed in the application, which are the Cryptographic Signature Analysis, the Heuristic Log Analysis and the Permission Analysis.

## 4.1 Cryptographic Signature Analysis

This section focuses on the concepts of cryptographic signatures and their subsequent analysis to detect malware. In the literature on this subject, fingerprints calculated with a hashing algorithm are often referred to as signatures.

There are currently other methods of malware detection, but the use of signatures or hash functions by comparing with the results of previously detected and analyzed malware is still the most functional technique for antivirus or security systems. Google Play Store requires that each APK must be signed with two digital certificates: an App signing key (used to sign APKs that are installed on a user's device) and an Upload key (used to sign the app bundle or APK before you upload it for app signing with Google Play). As part of Android security, the signing key never changes during the lifetime of an application, so it not only ensures that Android applications are trustworthy, but also verifies that the application has been provided by a trusted source [65]. If a third party manages to take an App signing key without the knowledge or permission of an app developer, it could sign and distribute the app that maliciously replaces the authentic application or corrupts it. Moreover, it could also sign and distribute apps under your identity that attack other apps or the system itself, or corrupt or steal user data.

The certificate fingerprint is a short and unique representation of a certificate that is often requested by API providers alongside the package name to register an app to use their service. The MD5, SHA-1 and SHA-256 fingerprints of the upload and app signing certificates can be found on the app signing page of the Play Console. When you are trying to publish an application you must have previously signed it by yourself providing the SHA-1 of your signing certificate or you upload it to the Play Console, and Play App Signing takes care of the rest. Google Play Store checks that the package name and certificate match with the application and if they do not match it is not offered to users but if it is an update of an existing application in the store it will consider it as a new application and will not offer it to users as an update [66].

The analysis of cryptographic signatures is based on database queries, which store the information obtained from previously reported or analyzed malicious files or applications. This information contains the summary functions of the malicious files which are used to uniquely and unambiguously identify each file hosted in the database.

Cryptographic signatures are a mathematical algorithm (hash function) that maps a data set, regardless of its size, to a bit-array of a fixed size. They are essential for malware detection

since in case of even the slightest modification of the data or code, the bit-array changes extensively. They are also deterministic so that a malicious file always generates the same bit array when applying the same hash function and if the hash function chosen has a weak collision, it would be impossible to find one malicious file and another valid file containing the same hash.

There are different types of algorithms used for hash function generation, but the most common are SHA2 (256, 384 or 512 bits), SHA1 (160 bits) and MD5 (128 bits). The notorious difference between the previously mentioned algorithms focuses on the length of the generated hash string, the longer the length of the string the lower the probability of a collision.

We can see in *Figure 28* an analysis of an Android application performed by the online tool VirusTotal [81], which makes use of a database management system that stores signatures. It focuses on performing file queries remotely using a hash function (SHA256) in order to check if the files are malicious.

The screenshot shows the VirusTotal analysis interface. At the top left is a green circular icon with a white '0' and '/ 60'. To its right is a message: 'No security vendors flagged this file as malicious' with a green checkmark icon. Below this are the file's SHA256 hash and name: 'fa5ec279b445df95ea9f54a41ab26009b988a8a0fce0cc3d7e44d91900881c09' and 'among-us.apk'. Underneath are tags: 'android' and 'apk'. Below the main panel are tabs: DETECTION, DETAILS (which is selected), RELATIONS, BEHAVIOR, and COMMUNITY. Under the DETAILS tab, there is a section titled 'Basic Properties' with a help icon. A table lists the following properties:

MD5	8f6deac483f667a85cf10442bf34292b
SHA-1	cec5ad5008a61e989b36cb356054f483964b7e6b
SHA-256	fa5ec279b445df95ea9f54a41ab26009b988a8a0fce0cc3d7e44d91900881c09
Vhash	e8dcc6687fe080858e142f6df3ee5bcf
SSDEEP	3145728:ovdEJrzOl6+cdrXkd5h+C3GidgjAwxbFFO9pHnHgUJQUSvG4:ovd4Or+c5XkZj3GidoFFO7nHgUJ6+4
TLSH	T165783393A6058E56C8F5553380D30B3666B39C116B120ADF8B1D7B3F34BEB476B22768
File type	Android

[81] Figure 28: VirusTotal - Analysis of an Android apk

## 4.2 Heuristic Log Analysis

Log files are computer-generated text files that are automatically produced whenever a specific event takes place in a specific environment, such as an operating system, application, server, etc. They contain information about usage, activities and operations. This information is useful for troubleshooting and debugging the environment, since they keep a record of everything that has happened in a textual format. They typically have the LOG file extension.

Each operating system has different methods of starting or stopping logs recording, since both the environment and the specific events that trigger them are different. Therefore, each OS is uniquely configured to generate log files in response to specific events. In the case of Linux, it divides log files into four categories: Application logs, Event logs, Service logs and System logs [67].

### 4.2.1 Android logs

Android Logging System consists of different circular buffers, which provide logging for different parts of the system. These log buffers are [68]:

- **radio:** This buffer contains radio/telephony related messages.
- **events:** This buffer stores binary system event messages.
- **main:** This is the default log buffer, which does not contain system and crash log messages (it contains the applications logs). This is the only buffer available to apps.
- **system:** This buffer contains the system logs.
- **crash:** This buffer stores logs related to crashes.
- **kernel:** This buffer stores kernel related logs.
- **security:** This is the security log buffer.
- **stats:** This buffer corresponds to statistics logs.

Each message in the log consists of a tag indicating the part of the system or application that the message came from, a timestamp, the message log level and the log message itself. The log level is a character that encodes the priority of the log entry (it is Android's terminology for severity level). Here we list all the possible values it can take, ordered from lowest to highest priority [68]:

- **V:** Verbose (lowest priority)
- **D:** Debug
- **I:** Info
- **W:** Warning
- **E:** Error
- **F:** Fatal
- **S:** Silent (highest priority, on which nothing is ever printed)

The Log class (`android.util.Log` [69]) is an API that allows users to create log entries based on their log level. It contains several public methods for logging in each priority. For

example, for Verbose priority the user can use the method `Log.v()`, for Warning priority the user can use `Log.w()`, etc. Typically, these methods take two arguments: the log's tag (for example, it could be the name of the activity that creates the log) and its message. The API then creates the log entry with the passed values and adds the timestamp, the identifier of the issuing process and thread and other information.

On the other hand, the Logcat command-line tool is used for reading logs. The user can run logcat through an adb shell using the following syntax:

```
[adb] logcat [<option>] ... [<filter-spec>] ...
```

This command has a wide variety of options; these are the most relevant [70]:

- **-b <buffer>**: Specifies the log buffer that is going to be read.
- **-c**: Clears the entire buffer.
- **-d**: Dump the log contents.
- **-v <format>**: Sets the output format for log messages. The default is threadtime format.

There are several output formats that modify the output so that they display certain metadata fields. The following list corresponds to the supported output formats [70]:

- **brief**: Display priority, tag, and PID of the issuing process.
- **long**: Display all metadata fields.
- **process**: Display PID only.
- **raw**: Display the raw log message with no other metadata fields.
- **tag**: Display the priority and tag only.
- **thread**: A legacy format that shows priority, PID, and TID of the thread issuing the message.
- **threadtime (default)**: Display the date, invocation time, priority, tag, PID, and TID of the thread issuing the message.
- **time**: Display the date, invocation time, priority, tag, and PID of the process issuing the message.

Now we show some examples of the most relevant formats:

- The **default output format** has the following structure:

```
Date Time PID TID Priority Tag: Message
```

```
05-16 20:04:58.151 6992 7560 i camera : open camera: 1, package  
name: com.whatsapp
```

- The **brief output format** has the following structure:

```
Priority/Tag( PID): Message
```

```
I/ActivityManager( 585): Starting activity: Intent {  
action=android.intent.action...}
```

- The **long output format** has the following structure:

```
[ Date Time PID: TID Priority/Tag ]  
Message  
[ 05-28 18:30:53.542 3716: 3733 I/com.whatsapp ]  
Background young concurrent copying GC freed 25395(1669KB)  
AllocSpace objects, 0(0B) LOS objects, 24% free, 6753KB/8917KB,  
paused 262us total 112.240ms
```

## 4.2.2 Log handling

Log files record a large amount of information that conveys everything that is happening in the system. This makes log files an important element to consider if we want to analyze what applications are running on the system and try to figure out what actions they are carrying out. Therefore, log analysis is a scrutiny method widely used in the industry of malware detection. The most typical use cases for log analysis are [71]:

- **Compliance** with security policies, audits or regulations.
- System **troubleshooting**.
- **Forensics**.
- Security **incident response**.
- Understanding online **user behaviour**.
- **Performance** improvement.

Depending on the use case, the behaviour of log analysis differs according to the context of the log files. After all, both the data and objective behind a network log analysis are not the same as a system log analysis. Hence, log analysis must interpret messages within the context of the application or system. Nonetheless, they usually have some procedures in common [71]:

- **Normalization:** Converting log messages from different sources into a uniform format.
- **Pattern recognition:** Selecting incoming log messages and comparing them with a previously established dataset to filter or handle the logs in different ways.
- **Classification and tagging:** Ordering and classifying log messages into different categories based on specific keywords, dates, etc for later usage.
- **Correlation analysis:** Collecting messages from different systems and finding all the messages belonging to one single event.
- **Artificial Ignorance:** Discarding log entries which are known to be uninteresting.

The normalization and classification procedures ensure an ease of use while handling the log messages. Secondly, the pattern recognition and correlation analysis procedures grant the analyst the information necessary to draw useful conclusions from the log entries. Lastly, the artificial ignorance procedure ensures the certainty of the results as well as a better performance.

Log analysis is a type of dynamic analysis, since it examines the behaviour of a system, network or applications while they are in execution. This implies it is a time and resources consuming process, since huge amounts of information are generated each second and

every log must be checked. Nevertheless, its greatest advantage is that it allows the administrator to discover how the analyzed element is interacting with the system, which helps discover malfunction or damages.

There are tools centred in Android log analysis to monitor system use. As an example, *SolarWinds Loggly* [72] has several functionalities that allow the user to perform an analysis of the logs of his device: it aggregates all of the Android logs on the cloud so that the user can monitor and analyze them by means of search queries and simplified charts and dashboards. Another example is *AndroidLogViewer* [73], which displays the logs of the user's Android device and allows him to search in them using regular expressions, filter them by tag, PID, priority, etc and more.

### 4.3 Permission Analysis

Android app permissions are considered to be a filter that helps to preserve user privacy by protecting access to restricted information, such as the user's system status and contact information, and to restricted actions, such as connecting to a linked device or recording audio. They lie in the *AndroidManifest.xml* file [24] and there are different classes depending on their purpose and restriction scope that they grant. This section covers the importance of Android permissions from a malware analysis approach and provides a detailed explanation of how they are assessed and classified.

Although the previous analysis and all the information declared in the *AndroidManifest* file must be considered as relevant, an Android permissions assessment is undoubtedly one of the sections to which more attention should be paid as a starting point when analyzing malware on Android. All system functionality that the application wants to access have to be declared within the *AndroidManifest.xml* file under the following tags structure:

```
<manifest>
    <uses-permission />
    <permission />
    <permission-group />
    ...
</manifest>
```

Since in this work the permission analysis is performed on the *<uses-permission />* tag, next, we review what its objective is.

#### **<uses-permission> tag**

This tag indicates what permissions an application requires, referring to hardware and software components that are on the device and that the application can make use of. From a malware analysis perspective, the key is to find some kind of unusual behaviours and other indicators. Therefore, in a study of declared permissions it is important to bare in mind the following tasks [74] as a guide:

- Identify those applications that request a large amount of permissions. These kinds of applications generally demand additional permissions without actually requiring them, which is a sign of unusual behaviour and might be a hint of a malware entry.
- Identify the functions that the application intends to perform through the declared permissions.
- Identify the permissions that it does not make sense to declare according to the supposed nature of the application. For example, an application whose supposed functionality is to allow the user to change the wallpaper desktop background, but which, through its permissions, requests sending of SMS messages.
- Identify the permissions that the application does not declare, but it would be expected to declare according to the supposed nature of the application. For example, a photography application that does not require access to the camera.
- Identify the permissions that the application declares and, according to its classification, look for those which are considered to be invasive and potentially dangerous. For example, permissions that delete packages, mount/unmount filesystems, read logs, etc.

On the other hand, if the application installation process is analyzed, some peculiarities should be observed depending on the type of installation that is carried out:

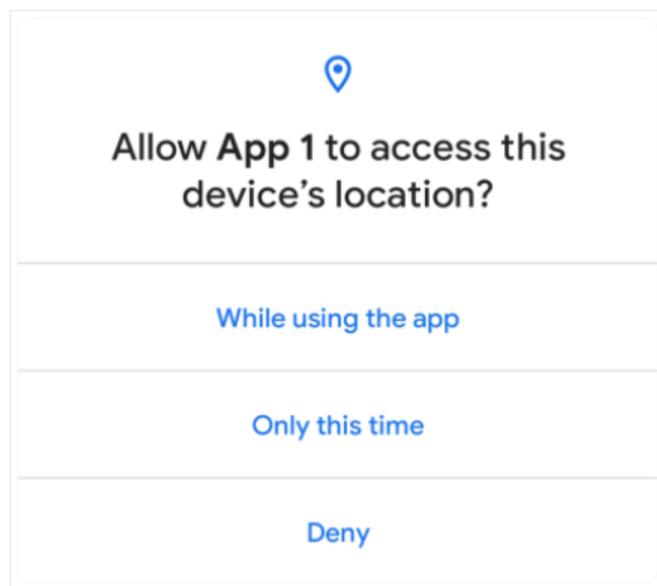
- In case of installing through the Google Play Store, the permissions are shown to the user grouped by categories to request their consent. At this point it is important to note a peculiarity that occurs when updating applications that are already installed on the device, since there may be threats that make use of it:
  - o If a permission from a category that has not been previously approved is added, a confirmation dialog is shown to the user in order to approve the new group of permissions.
  - o If a permission from a category that has been previously approved is added, no confirmation is requested from the user. So there might be the possibility that an application that initially requested a certain category, after an update, incorporates a new permission that belongs to the same category without asking the user for any confirmation. For example, an app that initially requests the *Messaging* category because it uses the `android.permission.READ_SMS` permission, after an update, it incorporates the `android.permission.WRITE_SMS` permission without asking the user for approval.
- In case of installing an APK through alternative markets such as Amazon AppStore or Aptoide (in any of the cases it is necessary to have allowed the installation from

unknown sources from the *Settings > Security* menu), all the requested permissions are shown to the user, even if it is an update of an installed application.

- In case of using other systems such as ADB [45] with its command `adb install app.apk` or by copying the APK file in one of the application directories, no approval is requested from the user with the permissions required by the application.

It is important to highlight that the application installation process, from the user's perspective, has changed in the current versions of the Android operating system (from Android Marshmallow 6.0 onwards). For the newest versions, several improvements have been introduced, allowing to define a higher level of granularity in the permissions and allowing the user to indicate which groups of permissions are going to be authorized [74].

From Android 6.0, the system offers the user the choice to change the permissions preferences by accessing *Settings > Applications > X App > Permissions* [74]. Now, with the release of the latest version (Android 11), the system gives users the ability to specify more granular permissions for location, microphone, and camera. Additionally, the system resets the permissions of unused apps that target Android 11 or higher, and apps might need to update the permissions that they declare if they use the system alert window or read information related to phone numbers. Furthermore, this Android version introduces the new permissions model known as *One-time permissions* [28]. Whenever an app requests a permission related to location, microphone, or camera, the user-facing permissions dialog contains an option called Only this time, as shown in *Figure 29*. If the user selects this option in the dialog, the app is granted a temporary one-time permission [75].



[75] Figure 29: System dialog that appears when an app requests a one-time permission.

# 5. Analysis Methods Implementation

This chapter gives a highly detailed and technical explanation of how we have applied the theory of chapter 4 for the development of each app analysis. The methods explained are used in the Android application, which is part of the results of this work, and other systems used in this project.

## 5.1 Cryptographic Signature Analysis Implementation

Malware on Android mobile platforms is a threat to the security of individuals and/or businesses. Through app republishing, apps are automatically downloaded and infected with malware, and then published under the name of the original app or under a slightly different one in both official and unofficial app shops. This is where the role of the hash function comes in, which guarantees the authenticity of the app by means of a unique bit array. The slightest change to the application would alter this array, resulting in a completely different one. This is one of the few effective methods of malware detection.

We first proceeded to implement a local database (MalwareDB) composed of three tables, as it can be seen in *Figure 30*.

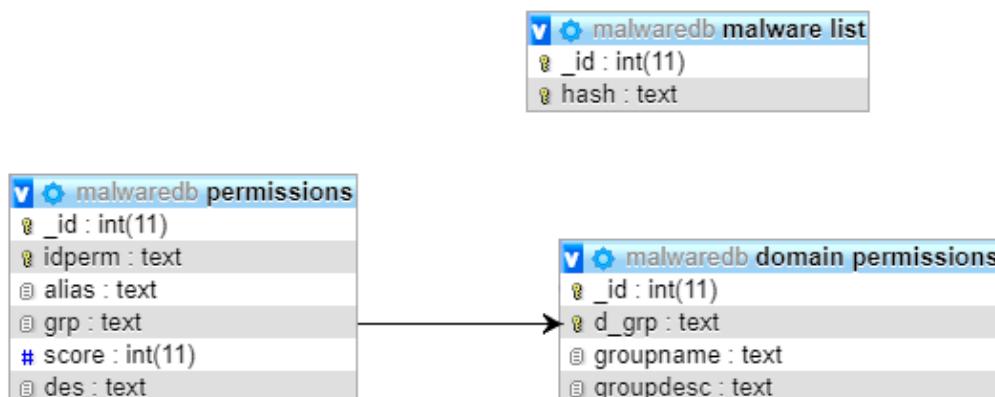


Figure 30: Entity Relationship Diagram (ERD) of Database used

Domain Permissions Table (see *Table 1*) groups permissions into domains and contains the following columns:

- **\_id**: Assigns a numerical identifier to organise each group of Android permissions.
- **d\_grp**: Attribute used for classifying each group of Android permissions.
- **groupdesc**: Contains a description of a specific group of Android permissions so that users are aware of its functionality and purpose of the permissions belonging to that group.

Domain Permissions		
Name	Type	Schema
_id	INTEGER	PRIMARY KEY AUTOINCREMENT
d_grp	TEXT	NOT NULL UNIQUE
		Group of one or more Android permissions under a particular category
groupname	TEXT	Name of group of permissions
groupdesc	TEXT	Description of group of permissions

Table 1: Domain Permissions Table structure

Malware List Table (see *Table 2*) contains a list of existing malware signatures and it is composed of:

- **\_id**: Assigns a numerical identifier to organise each malware hash.
- **hash**: Attribute used to be compared with an application hash that has been selected for analysis.

Malware List		
Name	Type	Schema
_id	INTEGER	PRIMARY KEY AUTOINCREMENT
hash	TEXT	NOT NULL UNIQUE
		MD5 hash function applied on malware

Table 2: Malware List Table structure

Permissions Table (see *Table 3*) has the totality of Android permissions and it is made up for:

- **\_id**: Assigns a numerical identifier to organise each Android permission.
- **idperm**: Attribute used for classifying each Android permission.
- **score**: Differentiates the type of permit (Forbidden, Dangerous, Deprecated, Signature, Normal, Unknown)
- **des**: Contains a description of a specific Android permission so that users are aware of its functionality and purpose.

Permissions		
Name	Type	Schema
_id	INTEGER	PRIMARY KEY AUTOINCREMENT
idperm	TEXT	NOT NULL UNIQUE
		Identifier of Android permission
alias	TEXT	Name given to Android permission
grp	TEXT	Foreign key grp with d_grp from table Domain Permissions
		Group of one or more Android permissions under a particular category
score	INTEGER	Grade of dangerousness of the permission
des	TEXT	Description of permission

Table 3: Permissions Table structure

In order to implement the cryptographic signature analysis we have to work with the Malware list table of the MalwareDB. For this project we have filled in that table thanks to the information from a repository of malware samples [76], provided by security researchers, incident responders, forensic analysts, and morbid curiosity. This repository contains a total of 38,598,420 malware samples grouped in lists consisting of one plain text file with one hash per line, from which we selected only MD5 list 8 as shown in *Figure 31*.

```
#####
# Malware sample MD5 list for #
# VirusShare_0008.zip          #
# http://VirusShare.com          #
# Twitter: @VXShare            #
#####
6ffa35b0a2acd5565ade6d3e1af64a94
12b5cc085c974ac955c37b1f84dda8ce
e9b2d68cd1de41c6278776f2d1249676
1c2a2d1853aafec963e5a62264f68134
11b8e669e71e956c138324272200e099
28ddb9b5b14fcc82c4b53af8ba0e03c
89dfd0590058f9021aac24e69a3132fe
5080cd5a0a56c99022a3fdfde6107f6b
e55afb50e57b8354102e37485bef0532
186b448e1ab099d6634c1c803b5d7286
e9870a5923a6465f1de82913057d067d
f60b8dd2d224bb692c14d0d8fcf933c4
a9fbfc4f3e537829bcb96289ab61273d
cab323903d20e7b9d6cd2142c7baf595
5d2296744d39432ccac99d9f3316c425
82cc3f74ed4594c393a3c882f6c31a64
33218d2de728bd921850b0bcf7056e1e
79334478d4b5bc02055fda4dd73103b4
```

[83] Figure 31: VirusShare - Malware sample MD5 hashes

The implementation of the cryptographic signature analysis is composed of five java classes (*Figure 32*) together with XML components linked to the graphic design that focuses on the use of the Android RecyclerView with a checkbox next to each item to display a list of installed apps on the device. We have used Android RecyclerView because not only does it efficiently display a list of large sets of data but also this reuse vastly improves performance, improving app's responsiveness and reducing power consumption.

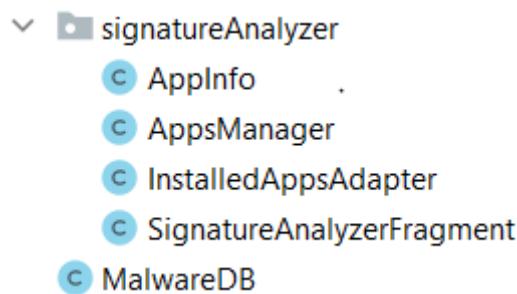


Figure 32: Structure and organisation of Java classes

Next, we explain each class that composes the Signature analyzer:

- **AppInfo:** This class (see *Figure 33*) is used to store specific information and particular values from an application such as the application's icon and name.

```

package com.example.androidmalwareanalyzer.ui.signatureAnalyzer;

import android.graphics.drawable.Drawable;

public class AppInfo {
    private String appName;
    private Drawable appIcon;
    private boolean isSelected;

    public Drawable getAppIcon() { return appIcon; }

    public void setAppIcon(Drawable appIcon) { this.appIcon = appIcon; }

    public boolean isSelected() { return isSelected; }

    public void setSelected(boolean selected) { isSelected = selected; }

    public String getAppName() { return appName; }

    public void setAppName(String appName) { this.appName = appName; }
}

```

Figure 33: Structure of AppInfo Java class

- **AppsManager:** This class works as a Package Manager by returning a list of all application packages that are installed for the user (*Figure 34*) by using `getInstalledApplications()`. Moreover, it is necessary to filter the system apps from the list of apps, for which we have used `ApplicationInfo.FLAG_SYSTEM`. Thus, just apps installed by the user in the device are obtained, owing to the fact that preinstalled apps are safe from malware. Furthermore, we wanted to grab the app name and icon from the `ApplicationInfo` class which retrieves information about a particular application installed in a device. In order to keep things easy and re-usable, we created two methods `setAppName()` and `setAppIcon()` that assign the icon and name of an existing application to a list of type `<AppInfo>`. The returned list is used to display on the screen all applications installed.

```

private void loadApps() {
    List<ApplicationInfo> packages = mContext.getPackageManager().getInstalledApplications( flags: 0);
    for (ApplicationInfo packageInfo : packages) {
        AppInfo newApp = new AppInfo();
        if ((packageInfo.flags & ApplicationInfo.FLAG_SYSTEM) == 0) {
            newApp.setAppName(getApplicationContextByPackageName(packageInfo.packageName));
            newApp.setAppIcon(getAppIconByPackageName(packageInfo.packageName));
            myApps.add(newApp);
        }
    }
}

```

Figure 34: Getting the list of installed applications

- **InstalledAppsAdapter**: It is a view adapter which controls how the Recyclerview shows the view. It also maps all application's information from the xml file to functions in the adapter. There is implemented a listener which wraps the entire item and if the checkbox is clicked, it is set to the opposite of what it was, so that with the use of `notifyDataSetChanged()` method, the list of apps is refreshed with its respective checkbox from RecyclerView.
- **SignatureAnalyzerFragment**: It is the main class where all button's functionalities are defined with their own listener and mapped to XML components based on the graphical view. Unfortunately, MD5 has been cryptographically broken and considered insecure. For this reason, it is always recommended to store cryptographic signatures using a different hashing algorithm, so we decided to implement a method (*Figure 35*) in which it examines the first line of a database added and it checks which hashing algorithm is used by its character length. This makes our app capable of updating the database management in case there is a hash database substitution.

```

String algorithm = dbHelper.getFirstHash();
algorithm = hashAlgorithms.get(algorithm.length());
String hash = getHashApp(app, algorithm);
public void InitializeHashAlgorithms(Map<Integer, String> hashAlgorithms) {
    hashAlgorithms.put(32, "MD5");
    hashAlgorithms.put(40, "SHA-1");
    hashAlgorithms.put(64, "SHA-256");
}

```

*Figure 35: Method to recognise which algorithm is used in the database*

## Problems overcome

The first version of the database presented a problem which was the repetition of the tuples when initialising our database, so to solve it we proceeded to modify hash, d\_grp and idperm columns to type UNIQUE.

Regarding the development of the cryptographic signature analyzer, several modifications were made until we reached the final version. After managing to display the list of applications installed on the Android device along with a checkbox, we decided to implement the SELECT ALL option to make it straightforward for the user to make a full selection of apps. In addition, it was decided to add a Searchbox as it was difficult to find a specific app among all the installed ones.

Being aware of the fact that the Malware List table was not secure enough in terms of the hashing algorithm used, we structured the cryptographic signature analyser so that it can be functional with any database containing a different hashing algorithm.

## 5.2 Heuristic Log Analysis Implementation

In the discussions carried out by the members of the team for the design of the malware detection tool, we came to the conclusion that a heuristic analysis of the log files was necessary. The reasons behind our decision to implement this analysis are the following:

- To alert the user of all the **applications** that are currently **running** on the device, since some of them may be running in the background without the user's knowledge.
- To let the user know what **target elements** (camera, SMS, storage, etc) each running application is trying to access.
- To let the user establish specific **keywords** that will be **monitored**. The result shows all the applications that shared a log entry with that specific keyword.
- To check if the applications that accessed certain target elements had the **permissions** necessary to do so.

Of all the procedures commonly used in log analysis, we use the following:

- **Classification and tagging:** We classify the logs by package name and keyword.
- **Correlation analysis:** We group all the logs of each application and correlate all the logs of a specific application by the target elements accessed.
- **Artificial Ignorance:** We only use those logs that contain something related to applications, since there are plenty of system logs that are not interesting for malware detection purposes.

We have developed this analysis dividing its functionality into two parts:

- **Application:** It connects the Android mobile device to the server, extracts its log entries, sends them to the server along with the package names of all the user apps installed on the device and some filters set by the user. After the analysis is stopped, it retrieves the result from the server and stores that result in the Previous Results database located in the Android device. Finally, it displays the conclusion visually, after processing the result.
- **Server:** Whenever an incoming connection from any mobile device arrives, it processes the logs received based on the filters established by the user and, after storing all the logs, it summarizes the entries of interest. Once the stop signal is received, it retrieves the results obtained and sends them back to the application installed in the mobile device through the connection.

### 5.2.1 Application

#### Setup

Reading logs is not something that a regular application should do. For this reason, the permission needed for accessing them, called `android.permission.READ_LOGS`, is a forbidden permission, which means that it cannot be granted by the user like normal permissions. One way of granting this type of permissions to applications is by rooting the device, which is extremely risky. Instead, the user can use ADB (Android Debug Bridge) [45] to open a shell that communicates with the device.

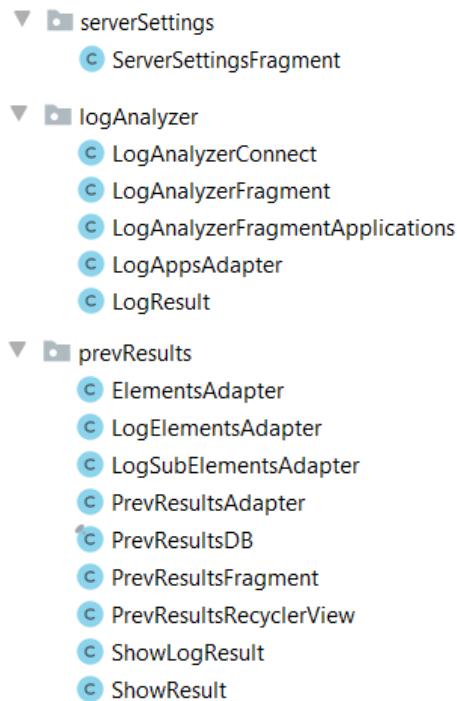
First of all, the user must be able to access the developer options of the Android device:

- This is achieved by clicking 7 times the build number, kernel version or other value inside the information about the phone (depends on the device).
- Inside the developer options the user must allow the USB debugging.
- The user needs to have installed ADB through *platform-tools* [82] on the computer to which the phone is going to be plugged.
- After plugging the phone to the computer and accepting to connect to said computer, the user must enter the directory of *platform-tools* through a terminal (CMD in Windows). Using the command `adb devices` we can check if the device is being detected. If it is, the user must execute the command `adb shell pm grant com.example.androidmalwareanalyzerandroid.permission.READ_LOGS`, which grants *AndroidMalwareAnalyzer* the *READ\_LOGS* permission.

## Classes developed

As *Figure 36* shows, three directories were developed to contain the functionality of the Log Analysis:

- **serverSettings:** Handles the IP and Port numbers used to connect to the server.
- **logAnalyzer:** All the functionality of the analysis is contained inside this directory, except saving the result obtained from the server and displaying it.
- **prevResults:** Contains the PrevResultsDB, which is the database used to store the results of both the signature and log analyses. It also contains the functionality needed for displaying said results.



*Figure 36: Organization of the Server Settings, Log Analyzer and Previous Results classes*

Next, we explain each class that composes the Server Settings:

- **ServerSettingsFragment:** This *Fragment* is used to handle the addresses (IP:Port) that the Log Analyzer will use to connect to the user. All the already stored addresses are shown through a *ListView*. These addresses can be selected by the user, simply by pressing them. The address selected by the user is the one that is going to be used for the connection. This *Fragment* also displays one button for adding addresses and one for deleting the selected address. The addresses are stored inside *SharedPreferences*. When the user adds a new address, both IP and Port numbers are parsed. The parsing method is shown in *Figure 37*.

```

try {
    String[] ip_port = newConnection.split( regex: "\\\\:");
    String[] parts = ip_port[0].split( regex: "\\\.");

    if (ip_port.length != 2)
        throw new IllegalArgumentException("Illegal argument. Only IP address and port needed.");

    if (parts.length != 4)
        throw new IllegalArgumentException("Illegal length of the IP address.");

    for (int i = 0; i < 4; i++) {
        try {
            int val = Integer.parseInt(parts[i]);
            if (val < 0 || val > 255) {
                throw new IllegalArgumentException("Illegal value '" + val + "' at byte " + (i + 1) + " in the IP address.");
            }
        } catch (NumberFormatException e) {
            throw new IllegalArgumentException("Illegal value '" + parts[i] + "' at byte " + (i + 1) + " in the IP address.");
        }
    }

    try {
        Integer.parseInt(ip_port[1]);
    } catch (NumberFormatException e) {
        throw new IllegalArgumentException("Illegal value '" + ip_port[1] + "' in the port number.");
    }
} catch (IllegalArgumentException e) {
    badAddress = true;
    Snackbar.make(view, e.getMessage(), Snackbar.LENGTH_LONG)
        .setAction( text: "Action", listener: null).show();
}
}

```

*Figure 37: ServerSettingsFragment - Address parsing*

Next, we explain each class that composes the Log Analyzer:

- **LogAnalyzerFragment:** This *Fragment* is the entrypoint of the log analysis. It is used for displaying two buttons. Both buttons redirect to the *LogAnalyzerFragmentApplications*, passing a boolean parameter, which tells the class if the applications to be listed have to be user applications (installed by the user) or system applications (stock-apps, system services). The user will have to choose between one of them.
- **LogAnalyzerFragmentApplications:** This *Fragment* is the frontend of the analyzer. It first creates a *RecyclerView* and sets its Adapter with a new class, called

*LogAppsAdapter*. Then, it also initializes the *LogAnalyzerConnect*. In this *Fragment*, the user can type some keywords that will be monitored while analyzing the logs. When the analysis is started, the *LogAnalyzerConnect* is executed. When the analysis is stopped, it saves in the *PrevResultsDB* the result obtained and calls the class *ShowLogResult* to display the result. This class also has an alert message system that, based on an integer, displays a *Snackbar* with information regarding the status of the analysis (*Figure 38*).

```

switch(status) {
    case 0:
        msg = "Select the applications to monitor.";
        break;
    case 1:
        msg = "Log monitoring started.";
        break;
    case 2:
        msg = "Log monitoring finished. Receiving results...";
        break;
    case 3:
        msg = "Data received.";
        break;
    case -1:
        msg = "No connections saved. Add a new connection in Server Settings.";
        break;
    case -2:
        msg = "No connections selected. Select a connection in Server Settings.";
        break;
    case -3:
        msg = "Unable to load the connections. Delete and create new ones in Server Settings.";
        break;
    case -4:
        msg = "Unable to connect to the socket. Check the connection in Server Settings.";
        break;
    case -5:
        msg = "Error produced while sending logs to the server.";
        break;
    case -6:
        msg = "Error produced while receiving logs from the server.";
        break;
    default:
}

```

*Figure 38: LogAnalyzerFragmentApplications - status*

- **LogAppsAdapter:** This class extends *RecyclerView.Adapter*. It is responsible for controlling the checkbox list of applications. It first retrieves all the applications installed on the device, both user and system apps (*Figure 39*). The package name, application name and icon of all the applications are stored in two *ArrayLists* of *PackageInfoStruct* classes, which we developed to store typical values of applications (*Figure 40*). One of the arrays contains the user apps and the other the system apps. Depending on whether the user chose to list user or system applications, the corresponding array is used in the checkbox list. For each application, this *Adapter* shows its icon and application name and a checkbox; if no application name is found, the package name is printed.

```

List<PackageInfo> packs = mContext.getPackageManager().getInstalledPackages( flags: 0);

for(int i = 0; i < packs.size(); ++i) {
    PackageInfo p = packs.get(i);
    PackageInfoStruct newInfo = new PackageInfoStruct();
    newInfo.setAppName(p.applicationInfo.loadLabel(mContext.getPackageManager()).toString());
    newInfo.setPackageName(p.packageName);
    newInfo.setAppIcon(p.applicationInfo.loadIcon(mContext.getPackageManager()));
    if ((p.applicationInfo.flags & ApplicationInfo.FLAG_SYSTEM) == 0)
        userApps.add(newInfo);
    else
        systemApps.add(newInfo);
}

```

Figure 39: LogAppsAdapter - get installed apps

```

public class PackageInfoStruct {
    private String appname;
    private String pname;
    private Drawable icon;
    private boolean isSelected;
    private String[] permissions;
    private String dir;
    private String version;
    private ApplicationInfo appinfo;
}

```

Figure 40: LogAppsAdapter - PackageInfoStruct

- **LogAnalyzerConnect**: This class extends *AsyncTask* [86]. It is responsible for extracting the logs, sending them to the server and retrieving the result from the server. It first loads the selected address that was stored in *SharedPreferences* in the *ServerSettingsFragment*. When this *AsyncTask* is executed, the *AsyncTask* method *doInBackground()* is called. This method first connects to the server using the loaded address (*Figure 41*). Once connected to the server, the logs are sent through the connection (*Figure 42*). When the user presses the button to stop the analysis, the result is retrieved through the connection (*Figure 43*). Finally, when *doInBackground()* finishes, the *AsyncTask* method *onPostExecute()* sends the result back to *LogAnalyzerFragmentApplications*.

```

address = new InetSocketAddress(ip_server, port_server);
socket = new Socket();
try {
    socket.connect(address);
} catch (IOException e) {
    e.printStackTrace();
    return -4;
}
ret = 0;

```

Figure 41: LogAnalyzerConnect - New connection

```

while(!stop) {
    Process process = Runtime.getRuntime().exec(new String[]{"logcat", "-d"});
    Runtime.getRuntime().exec(new String[]{"logcat", "-c"});
    BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(process.getInputStream()));

    while ((line = bufferedReader.readLine()) != null) {
        if (line.charAt(0) != '-') {
            line += '\n';
            ostream.write(line.getBytes());
        }
    }

    Thread.sleep( millis: 3000);
}

```

Figure 42: LogAnalyzerConnect - Extract and send logs

```

InputStream istream = socket.getInputStream();
BufferedReader in = new BufferedReader(new InputStreamReader(istream));

while(!(line = in.readLine()).equals("Q")) {
    msg += line + '\n';
}

socket.close();

```

Figure 43: LogAnalyzerConnect - Retrieve result

- **LogResult:** This class is responsible for converting the result from text into an array of *LogInteractions* classes (Figure 44), which we developed to store information needed to display the result. It also has a function that reduces the size of the result, as the data retrieved from the server may have some application results split over more than one line, allowing it to be reduced to a single line. This way, the database saves some space.

```

public class LogResult {
    private ArrayList<LogInteractions> logs_result_list;

    public class LogInteractions implements Comparable<LogInteractions> {
        private String element = "";
        private String appname = "";
        private Drawable icon;
        private int total_interactions = 0;
        private ArrayList<SubElements> sub_elements;
        private boolean is_expanded = false;

    }

    public class SubElements implements Comparable<SubElements> {
        private String element = "";
        private String appname = "";
        private Drawable icon;
        private int interactions = 0;
        private boolean permission_granted = false;
    }
}

```

Figure 44: LogResult - Attributes

Next, we explain each class that composes the Previous Results:

- **PrevResultsDB:** This class acts as a database for the results obtained from the log and signature analysis methods. This database contains a table called *prevResults*, which is formed by the columns shown in *Table 4*. The datetime of the completion of the analysis is stored to know when the analysis was performed. The *analysis\_type* row differentiates between the signature and log methods. The *apps\_analysed* value is used to know what elements were analyzed and the *analysis\_result* stores the result obtained from the analysis. The two main methods of this database are *insertToDB()* (inserts a new result into the table) and *readAllFromDB()* (retrieves all the results performed).

PrevResults		
Column name	Type	Schema
<code>_id</code>	INTEGER	PRIMARY KEY AUTOINCREMENT
<code>date</code>	TEXT	Date and time when the analysis was finished
<code>analysis_type</code>	TEXT	Describes the analysis type (log or signature)
<code>apps_analysed</code>	TEXT	List of the keywords and package names of the applications analyzed
<code>analysis_result</code>	TEXT	Result of the analysis

*Table 4: Previous Result Table structure*

- **PrevResultsFragment:** this *Fragment* is the entrypoint of the Previous Results. It first gets all the results stored in the database through the method *readAllFromDB()* and displays them in a *RecyclerView* whose *Adapter* is an instance of *PrevResultsAdapter*. If the user selects one of the elements of the list, she/he will be redirected to *ShowResult* if it is a signature analysis result or *ShowLogResult* if it is a log analysis result. If no analysis has been performed, the message "No analysis performed yet" is displayed.
- **PrevResultsAdapter:** This class extends *RecyclerView.Adapter*. For each element, it displays the date when the analysis was performed and the analysis type (signature or log).
- **ShowResult:** This *Fragment* displays the result obtained from a signature analysis. Initially, this class calls its method *getInstalledApps()* (see *Figure 45*). First of all, this method checks if no applications have been analyzed. If true, then the list of apps analyzed will display "None". If at least one application has been analyzed, it gets the application name, package name and application icon of the analyzed

applications. This process is similar to the one used in *LogAppsAdapter*. These applications are then displayed using the *PrevResultsRecyclerView* class, with the *ElementsAdapter*. Afterwards, a *TextView* displays the result.

```

if (filter_values.length < 1 || elements.equals("")) {
    PackageInfoStruct newInfo = new PackageInfoStruct();
    newInfo.setAppName("None");
    newInfo.setAppIcon(null);
    elements_analyzed.add(newInfo);
}
else {
    for (int i = 0; i < filter_values.length; i++) {
        found = false;
        PackageInfoStruct newInfo = new PackageInfoStruct();

        for (int j = 0; j < packs.size() && !found; j++) {
            PackageInfo p = packs.get(j);
            if (filter_values[i].equals(p.packageName)) {
                newInfo.setAppName(p.applicationInfo.loadLabel(mContext.getPackageManager()))
                newInfo.setPackageName(p.packageName);
                newInfo.setAppIcon(p.applicationInfo.loadIcon(mContext.getPackageManager()))
                found = true;
            }
        }

        if (!found)
            newInfo.setAppName(filter_values[i]);
    }

    elements_analyzed.add(newInfo);
}

```

Figure 45: *ShowResult - getInstalledApps*

- **PrevResultsRecyclerView:** This class extends *RecyclerView*. It was developed to establish a dynamic size for the list, since the list had to have the necessary height to wrap its contents, but also maximum height, so that it did not take up too much space. As seen in *Figure 46*, this was accomplished by overriding the *onMeasure()* method and setting a height of 750 at most.

```

@Override
protected void onMeasure(int widthSpec, int heightSpec) {
    heightSpec = View.MeasureSpec.makeMeasureSpec(size: 750, View.MeasureSpec.AT_MOST);
    super.onMeasure(widthSpec, heightSpec);
}

```

Figure 46: *PrevResultsRecyclerView - onMeasure*

- **ElementsAdapter:** This class extends *RecyclerView.Adapter*. It displays all the applications or elements analyzed (the keywords specified in *LogAnalyzerFragmentApplications*).

- **ShowLogResult:** This *Fragment* is used to show the results of a log analysis. It firstly creates an instance of *LogResult* and calls its method *getList()*, which converts the result from text to an array of *LogInteractions*. Afterwards, the method *getInstalledApps()* of *ShowLogResult* is called (see *Figures 47 and 48*). The functionality of this method is quite similar to the also named *getInstalledApps()* method of *ShowResult*. First, if no elements were analyzed, an *ArrayList* is created with only one element that will be displayed as “*Everything*”, since not choosing any element to analyze implies analyzing all the user applications. If at least one element was analyzed, the *ArrayList* is set with the application name, package name and application icon of the analyzed apps. Then, it traverses all the elements and sub elements present in the result, adding the application name, package name and application icon of the applications and the name of keywords and target elements to two *ArrayLists* of *LogInteractions*. The elements of the result can either be applications (which implies that their sub elements are the target elements they have accessed) or keywords set by the user (which implies that their sub elements are the applications that have accessed that keyword). Having the three *ArrayLists*, first the elements analyzed (apps and keywords) are displayed through a *PrevResultsRecyclerView*, with the *ElementsAdapter*. Then, a list of all the applications that were running during the analysis is shown in a *RecyclerView*, with the *LogElementsAdapter* for the elements (applications) and *LogSubElementsAdapter* for the sub elements (target elements) since the applications that have accessed at least one target element can be expanded. Next, a list of the keywords specified by the user and that were found during the analysis is shown in a *RecyclerView*, with the *LogElementsAdapter* for the elements (keywords) and *LogSubElementsAdapter* for the sub elements (applications) since the keywords that have been accessed by one application can be expanded. Finally, the *setPermissionsList()* method checks if the applications that have accessed a target element had the permissions necessary to do so (see *Figure 49*) (we decided it would be better to check whether the apps have a permission within a specific permission group instead of comparing all possible permissions, because the permissions needed are well summarized by their permission group; the only exception is *NFC*, since it is a very specific permission inside the permission group *Network*, which is extremely broad). This is achieved by verifying all the permissions granted to those applications (see *Figure 50*) and contrasting if one of those permissions allows the app to access the specific target element. For this purpose a list with the target elements and their corresponding necessary permissions () has been created to check if the applications have them granted; the values are displayed in *Table 5*. The result of checking the permissions is shown in a *RecyclerView*, with the *LogElementsAdapter* for the elements (apps / keywords) and *LogSubElementsAdapter* for the sub elements (target elements / apps) since the applications that have accessed at least one target element can be expanded. A symbol at the right of each element shows if the necessary permission is granted (green check) or not (red cross).

```

List<PackageInfo> packs = mContext.getPackageManager().getInstalledPackages( flags: 0);
String[] filter_values = elements.split( regex: ",");
boolean found;
elements_analyzed = new ArrayList<PackageInfoStruct>();

//Elements analyzed
if (filter_values.length < 1 || elements.equals("")) {
    PackageInfoStruct newInfo = new PackageInfoStruct();
    newInfo.setAppName("Everything");
    newInfo.setAppIcon(null);
    elements_analyzed.add(newInfo);
}
else {
    for (int i = 0; i < filter_values.length; ++i) {
        found = false;
        PackageInfoStruct newInfo = new PackageInfoStruct();

        for (int j = 0; j < packs.size() && !found; ++j) {
            PackageInfo p = packs.get(j);
            if (filter_values[i].equals(p.packageName)) {
                newInfo.setAppName(p.applicationInfo.loadLabel(mContext.getPackageManager()).toString());
                newInfo.setPackageName(p.packageName);
                newInfo.setAppIcon(p.applicationInfo.loadIcon(mContext.getPackageManager()));
                found = true;
            }
        }

        if (!found)
            newInfo.setAppName(filter_values[i]);
        elements_analyzed.add(newInfo);
    }
}

```

Figure 47: ShowLogResult - getInstalledApps1

```

for (int j = 0; j < packs.size(); ++j) {
    PackageInfo p = packs.get(j);

    for (int i = 0; i < logs_result_list.size(); ++i) {
        if (logs_result_list.get(i).getElement().equals(p.packageName)) {
            logs_result_list.get(i).setAppName(p.applicationInfo.loadLabel(mContext.getPackageManager()).toString());
            logs_result_list.get(i).setAppIcon(p.applicationInfo.loadIcon(mContext.getPackageManager()));

            logs_result_app_list.add(logs_result_list.get(i));
        }
    }

    found = false;

    for (int k = 0; k < logs_result_list.get(i).getSubElementsSize() && !found; ++k) {
        if (logs_result_list.get(i).getSubElement(k).equals(p.packageName)) {
            found = true;
            logs_result_list.get(i).setSubElementApp(k, p.applicationInfo.loadLabel(mContext.getPackageManager()), p.applicationInfo.loadIcon(mContext.getPackageManager()));
        }
    }
}

for (int i = 0; i < logs_result_list.size(); ++i) {
    found = false;

    for (int j = 0; j < logs_result_app_list.size(); ++j) {
        if (logs_result_list.get(i).getElement().equals(logs_result_app_list.get(j).getElement()))
            found = true;
    }

    if (!found)
        logs_result_keyword_list.add(logs_result_list.get(i));
}

```

Figure 48: ShowLogResult - getInstalledApps2

```

List<Permission> permissions_needed = generatePermissionList();

for (int i = 0; i < logs_result_app_list.size(); ++i) {
    List<String> granted_permissions = getGrantedPermissions(logs_result_app_list.get(i).getElement());

    for (int k = 0; k < logs_result_app_list.get(i).getSubElementsSize(); ++k) {
        found = false;

        for (int q = 0; q < permissions_needed.size() && !found; ++q) {
            if (logs_result_app_list.get(i).getSubElement(k).equals(permissions_needed.get(q).getElement_name())) {
                found = true;
                found2 = false;

                for (int j = 0; j < granted_permissions.size() && !found2; ++j) {
                    PermissionFragment.PermissionInformation perm = dbHelper.getPermission(granted_permissions.get(j));
                    if (!permissions_needed.get(q).getDomain_name().equals("")) {
                        if (permissions_needed.get(q).getDomain_name().equals(perm.getDomain_id())) {
                            found2 = true;
                            logs_result_app_list.get(i).setSubElementPermissionGranted(k, status: true);
                        }
                    } else {
                        if (permissions_needed.get(q).getPermission_name().equals(perm.getPerm_id())) {
                            found2 = true;
                            logs_result_app_list.get(i).setSubElementPermissionGranted(k, status: true);
                        }
                    }
                }
            }
        }
    }

    if (logs_result_app_list.get(i).getSubElementsSize() > 0)
        logs_result_permission_list.add(logs_result_app_list.get(i));
}

```

Figure 49: ShowLogResult - setPermissionsList

```

    PackageInfo pi = mContext.getPackageManager().getPackageInfo(appPackage, PackageManager.GET_PERMISSIONS);
    for (int i = 0; i < pi.requestedPermissions.length; i++) {
        if ((pi.requestedPermissionsFlags[i] & PackageManager.REQUESTED_PERMISSION_GRANTED) != 0) {
            granted.add(pi.requestedPermissions[i]);
        }
    }
}

```

Figure 50: ShowLogResult - getGrantedPermissions

Target elements and permissions needed		
Target Element	Permission	Permission Group
SMS/MMS	-	MESSAGES
Location	-	LOCATION
Camera	-	CAMERA
Microphone	-	MICROPHONE
Telephone	-	PHONE_CALLS
Bluetooth	-	BLUETOOTH_NETWORK
Internet	-	NETWORK
Messaging	-	NETWORK
Storage	-	STORAGE
SDcard	-	STORAGE
Contacts	-	SOCIAL_INFO
NFC	NFC	-
Mail	-	MESSAGES
Accounts	-	ACCOUNTS

Table 5: Target elements and permissions needed

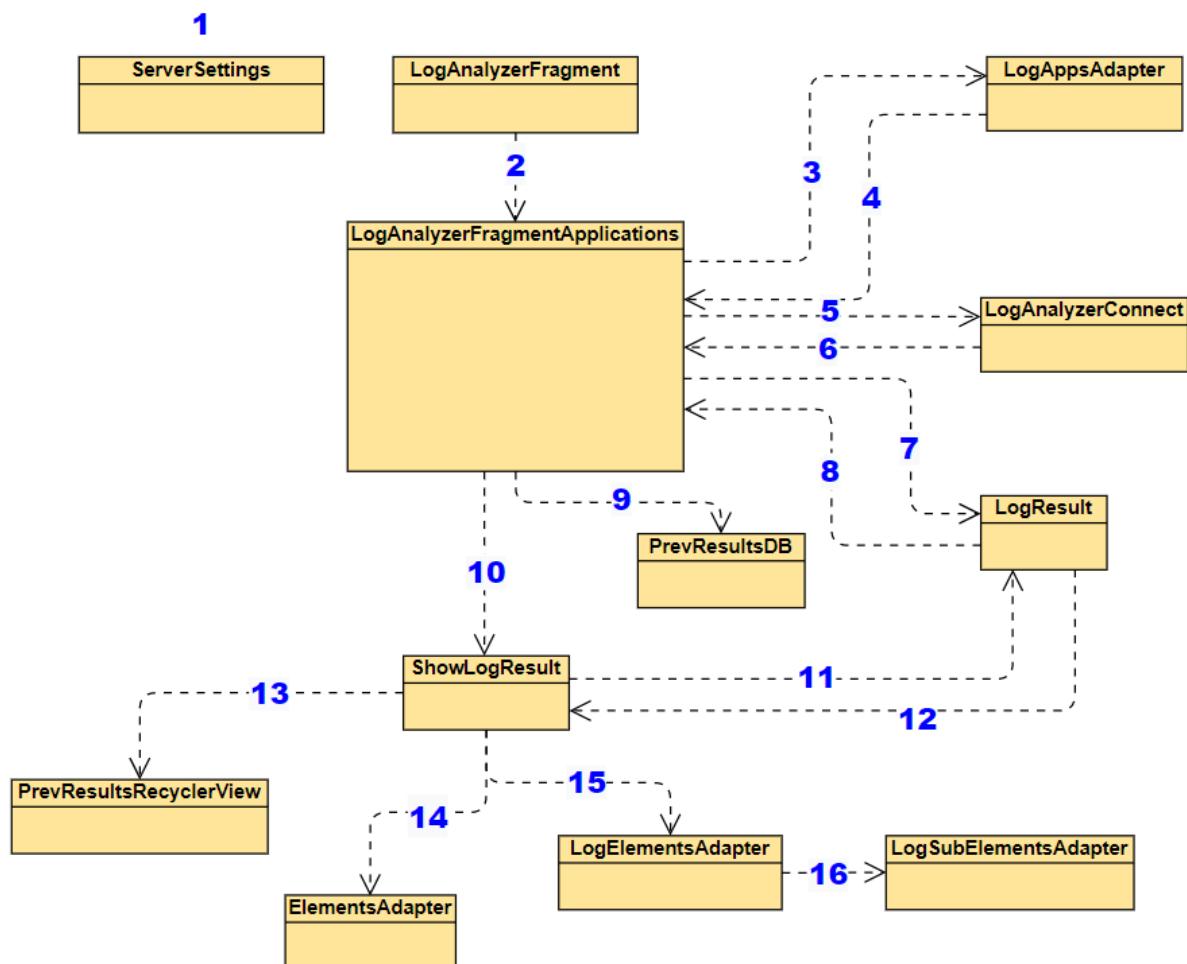
- **LogElementsAdapter:** This class extends *RecyclerView.Adapter*. For each element, it displays an icon, a name and the number of log entries they have appeared in (the permissions change this number for a symbol that encodes if that permission is granted). If the element is an application, the icon and names used are those of the application. If the element is a keyword that corresponds to a target element, an icon fitting the target element is used, and the name displayed is the target element. In case the keyword does not correspond to a target element, a default image is printed

alongside the keyword. In case the element has sub elements, it can be expanded. This shows another *RecyclerView* inside, which uses the *LogSubElementsAdapter*. Applications can have target elements as their sub elements and keywords can have applications as their sub elements.

- **LogSubElementsAdapter:** This class extends *RecyclerView.Adapter*. For each element, it displays an icon, a name and the number of log entries they have appeared in (the permissions change this number for a symbol that encodes if that permission is granted). If the element is an application, the icon and names used are those of the application. If the element is a target element, an icon fitting the target element is used, and the name displayed is the target element.

## Class diagram

This section explains how the classes involved in the log analysis communicate with each other. *Figure 51* has been designed to help the reader visualize the flow of classes. Each communication has been given a number, ordered chronologically. Afterwards, each indexed communication is explained.



*Figure 51: Connections between classes*

1. The application first needs to have knowledge of the IP and Port numbers that are going to be used to connect to the server. For this reason, the user must first go to the *ServerSettingsFragment* and add the address of the server.
2. When designing the user's interaction with the log analysis part of the system, it is important to first know which applications the user wants to analyze. To achieve this, the user must decide between two buttons once she/he has entered the *LogAnalyzerFragment*. Pressing the first button implies that the user wants to analyze the apps she/he installed; the other one is used when the user wants to monitor system applications. After one of the buttons has been pressed, the app redirects to *LogAnalyzerFragmentApplications*, passing a boolean variable that encodes the decision taken by the user.
3. *LogAnalyzerFragmentApplications* initializes *LogAppsAdapter*, passing the boolean variable received in communication 2, which is used to decide what type of applications are going to be listed.
4. When the user presses the *Start Analysis* button, a string containing all the package names of the applications selected by the user is passed back to *LogAnalyzerFragmentApplications*.
5. *LogAnalyzerFragmentApplications* executes *LogAnalyzerConnect*, passing the string received in communication 4, a string containing the keywords typed by the user and a string containing all the package names of all the user applications installed in the device.
6. Once the connection to the server has been established, the log entries extracted get sent to the server. When the user presses the *Stop Analysis* button, the result is received from the server as a string, which is passed back to *LogAnalyzerFragmentApplications*.
7. The result string is passed to *LogResult*, which transforms the result from a string into an *ArrayList* of *LogInteractions* classes.
8. Then, the array is converted back into a string. By doing this, the size of the result is reduced since the data retrieved from the server may have some application results split over more than one line, allowing it to be reduced to a single line.
9. The result and its metadata are inserted into *PrevResultsDB*. The values stored are the type of result (in this case “*Log analysis result*”), the current date, the analyzed application packages and keywords and the reduced result obtained from communication 8.
10. The application then redirects to *ShowLogResult*. The arguments passed to this class are the same as those passed in communication 9, but adding the *Context* as well.
11. A new *LogResult* is created. It receives the result in string format and transforms it into an *ArrayList* of *LogInteractions* classes.

12. The *ArrayList* obtained is returned to *ShowLogResult*.
13. *PrevResultsRecyclerView* is used for limiting the height of the *RecyclerView*. It does not receive any arguments.
14. *ElementsAdapter* receives the analyzed applications and keywords and displays them in the *RecyclerView* created in communication 13.
15. *LogElementsAdapter* receives an *ArrayList* of *LogInteractions* containing the elements that are going to be displayed, a boolean value that will tell the class if the list should have the permission format and the *Context*. If it has the permission format, a symbol will be displayed on the right side of the element; otherwise, the number of log entries for the corresponding element is displayed.
16. *LogSubElementsAdapter* receives an *ArrayList* of *LogInteractions.SubElements* containing the elements that are going to be displayed, a boolean value that will tell the class if the list should have the permission format and the *Context*. If it has the permission format, a symbol will be displayed on the right side of the element; otherwise, the number of log entries for the corresponding element is displayed.

### 5.2.2 Server

When implementing the log analysis, we have decided to apply an application server configuration; having explained the application features, we describe in this subsection the implementation of the server side.

#### Setup

In our case we decided to use AWS EC2 for hosting the server, but the instructions given can be used for any cloud service. First, the user must create a Linux instance, making sure that the TCP connections are allowed. More specifically, connections from ports 22 (used to connect to a terminal through ssh) and 1234 (used to receive the logs from the application; we decided to use this port number, but can be any other) must be allowed. Once this instance is running and a ssh connection to it has been established, some installations must be done. Then, some configuration must be performed.

Install Java:

```
sudo apt-add-repository ppa:webupd8team/java
sudo apt-get update
sudo apt install openjdk-8-jdk
```

Install Scala:

```
sudo apt-get install scala
```

Install Python:

```
sudo apt-get install python
```

Install Spark:

```

sudo curl -O \\
http://d3kbcqa49mib13.cloudfront.net/spark-2.2.0-bin-hadoop2.7.tgz
sudo tar xvf ./spark-2.2.0-bin-hadoop2.7.tgz
sudo mkdir /usr/local/spark
sudo cp -r spark-2.2.0-bin-hadoop2.7/* /usr/local/spark

```

Add `/usr/local/spark/bin` to the PATH:

```
export PATH="$PATH:/usr/local/spark/bin"
```

Include the internal hostname and IP to `/etc/hosts`. For example:

```

127.0.0.1 localhost
172.30.4.210 ip-172-30-4-210

```

Finally, download the two necessary scripts that we have developed for the server:

- **server.c**: Manages incoming connections, parses incoming logs, sends them to the `analyzer.py` script and, when the user has stopped the analysis, it gives the result back to the user.
- **analyzer.py**: Spark Streaming script that stores, splits and reduces the logs.

To execute the scripts, open two terminals. In the first one execute:

```

gcc -Wall -g server.c -o server
./server

```

In the other one execute:

```
spark-submit analyzer.py
```

Next, the functionality of both scripts (`server.c` and `analyzer.py`) is going to be explained in detail.

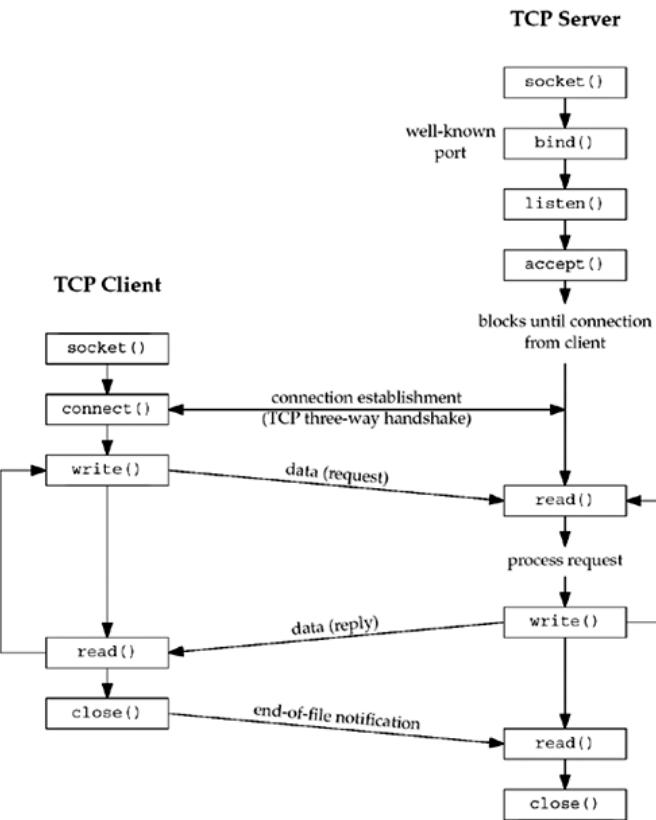
## Server.c

The `server.c` script performs three functions described below:

### 1. Connection management

First, the script waits until it can connect with `analyzer.py` via TCP. The socket established for this connection is identified by the address `localhost` and the port `9999`. Once the connection with the script has been established, the server begins to accept client connections.

We have programmed this script to always be in execution, allowing all clients to connect to the server whenever they need. For this reason, the server has been developed as a concurrent server with an accept-and-fork pattern. We can see this pattern in *Figure 52*, but after the method `accept()` returns, the server forks.



[85] Figure 52: Elementary TCP Socket

When the connection has been established, the function `receiveData()` is called.

## 2. Log management

The logs are managed inside the method `receiveData()`. This method first receives the keywords and applications that are used as filters and the package name of all the user applications of the client's device.

Then, the process loops until the client sends a specific signal that instructs the server that the user has finished sending logs. For ease of use, we decided to lower case all the logs received.

Now, the server checks if the logs contain any of the package names from among all the user applications on the client device, which were received at the start of this method. If one is found, we consider that log to be of interest, since we only want to monitor the behaviour of the user applications.

Since we want to show the user what applications are running, we send the logs related to the target elements to be monitored to the `analyzer.py` script, concatenating at the beginning of the log the IP of the device of the client and the package name of the application found within the log.

We decided that there are several target elements that require special attention in terms of possible malicious behavior. These target elements are:

- location
- gps
- camera
- microphone
- sound
- recorder
- telephony
- bluetooth
- wifi
- network
- messaging
- mms
- sms
- sdcard
- storage
- contacts
- nfc
- mail
- account

Now, three checks are performed:

- **The keywords and applications filters are empty:** we check if there are any target elements in the log. If there is one found, we send it to *analyzer.py*, concatenating the client's device IP, the package name of the application found inside the log and that target element.
- **The keywords filter is not empty:** we check if there are any keywords of the keywords filter in the log. If there is one found, we send it to *analyzer.py*, concatenating the client's device IP, the keyword and the package name of the application found inside the log.
- **The applications filter is not empty:** we check if the application is in the specified applications filter. If it is, we check if there are any target elements in the log. If there is one found, we send it to *analyzer.py*, concatenating the client's device IP, the package name of the application found inside the log and that target element.

### 3. Getting the result

Since spark streaming stores everything in several partitions, we created the function *getResults()*, which opens every file inside every subdirectory found in the *Result* directory. Then, we send to the client all the results that contain his IP and delete those results.

*Analyzer.py*

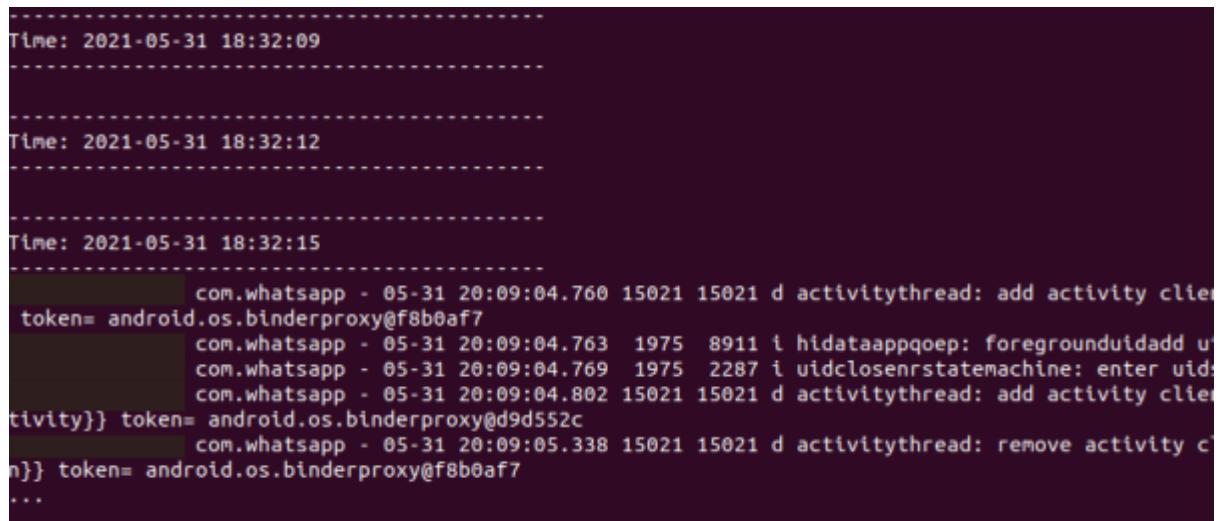
We have programmed this script to always be in execution, allowing all clients to connect to the server whenever they need. Figure 53 shows the code of this script.

```
if __name__ == "__main__":
    sc = SparkContext("local[2]", appName="Logs")
    ssc = StreamingContext(sc, 3)
    sc.setLogLevel("ERROR")
    lines = ssc.socketTextStream("localhost", 9999)
    RDD_lines = lines.filter(lambda line: line != "")
    RDD_lines.pprint(5)
    RDD_lines.saveAsTextFiles("AndroidLogs/") #save all logs
    if(RDD_lines.count() != 0):
        rdd_split = RDD_lines.map(lambda x: x.split())
        res = rdd_split.map(lambda l: ((l[0], l[1], l[2]), 1)).reduceByKey(add) #((172.0.0.1, app.com.example, camera), 1)
        res.saveAsTextFiles("Result/")
    ssc.start()
    ssc.awaitTermination()
```

Figure 53: analyser.py

First, the script waits until it can connect with the *server.c* via TCP. The socket established for this connection is identified by the address *localhost* and the port *9999*. Once the connection with the script has been established, the server begins to accept client connections.

It is a Spark Streaming script, which implies that it is constantly getting information. First, it saves the logs received in the *AndroidLogs* directory. Then, it parses those logs and reduces them with the following format ((IP, package name, target element), number of grouped logs). Finally, it saves those reductions in the *Result* directory. Figure 54 shows a snippet of the script in execution.



```
Time: 2021-05-31 18:32:09
-----
Time: 2021-05-31 18:32:12
-----
Time: 2021-05-31 18:32:15
    com.whatsapp - 05-31 20:09:04.760 15021 15021 d activitythread: add activity client
    token= android.os.binderproxy@f8b0af7
    com.whatsapp - 05-31 20:09:04.763 1975 8911 i hidataappqoep: foregrounduidadd u
    com.whatsapp - 05-31 20:09:04.769 1975 2287 i uidclosenrstate: enter uid
    com.whatsapp - 05-31 20:09:04.802 15021 15021 d activitythread: add activity client
    activity}} token= android.os.binderproxy@d9d552c
    com.whatsapp - 05-31 20:09:05.338 15021 15021 d activitythread: remove activity c
    n}} token= android.os.binderproxy@f8b0af7
...
```

Figure 54: analyzer.py running

### 5.2.3 Problems overcome

- The IP and Port numbers of the server were initially hard-coded. The problem of this approach is that the IP of the EC2 instance changes each time the instance is launched, which implies that the application had to be modified, compiled and installed whenever the cloud instance was launched. For this reason, we decided to implement a TextView where the user could write the IP and Port. We then realized that every time the application was launched, the user had to input the IP and Port, which was a very exhausting task. Finally, we opted to develop the *ServerSettings* class for storing the IP and Ports. With this class, the user has to input them only once.
- At first, all types of applications were printed (user and system apps). This made it too difficult for the user to choose a specific application, so we decided to divide them into two different fragments. We also decided to keep the system applications because it can be very interesting when performing a log analysis to analyze by Camera, Bluetooth, etc.
- In Android, an application cannot connect to a socket in the UI Thread (main thread of execution for the application). This meant that the connection to the server had to be executed in another thread. We decided to achieve this by using the *AsyncTask* class for its simplicity in programming and the fact that its computation runs in a background thread and its result is posted to the UI thread.
- When sending data to the server, we realized that `recv()` call did not always read all the data sent. More specifically, if the size of the data was too big, only a fraction of that data was received. We at first thought that it could be due to the size of the buffer where the data was being stored, but we later realized that we shouldn't expect to receive the data in the same number of read calls as there were write calls. For that reason, we used termination characters, to specify the end of the data being sent. We used '#' for the user app package names, '\n' for the logs and 'Q' for signaling the end of the communication.
- At first we performed the log filtering in the application. A problem arose, which was that some devices could not handle such a large amount of processing. For that reason, we decided to move the log filtering to the server. This also allowed us to store all the logs in the server before doing any filtering.
- One of the biggest challenges we faced was how to extract useful information from the logs. The main problem with logs is that they share a shared format, but the message itself does not have a common form, so each message is different. We first thought that it would be interesting to know which applications created a log entry with a level above Warning, because that could reveal applications with bugs in the code or applications that access specific protected items. In the end, we decided that the best way was to check for specific patterns inside those messages, and that's how we came up with the idea of the target elements.

- We also faced some issues returning the result from *LogAnalyzerConnect* to *LogAnalyzerFragmentApplications*, as it runs on a different thread. We managed this by creating a delegate function in the UI thread and passing it to *LogAnalyzerConnect*. This function acts as an asynchronous response that is called within *AsyncTask*'s *onPostExecute()* method, which is called when all processing has already been done.

## 5.3 Permission Analysis Implementation

Another aspect that must be considered in the development of a malware analysis tool is the study of the relevance of the permissions requested by each application. Permissions play a very important role when analyzing malware as they help support user privacy by protecting access to restricted data and restricted actions from malicious purposes [55]. This section goes through the motivations behind performing this assessment and explains all the details and steps followed during the implementation of the permission analysis.

The permissions determine what is allowed to be done by an app. In order to perform a more comprehensive and elaborate malware study, the team has concluded that a permission analysis would make a solid supporting feature to our Android Malware Analyzer App. To carry out an analysis of this style, the main thing is to remain neutral and always rely on objective facts to achieve the most accurate result possible in the permission assessment. The fact of fulfilling these conditions determines the rigor and precision of the analysis.

The basis and what we are trying to evaluate in this analysis are the permissions requested by the application, which can be found inside the *AndroidManifest.xml* file [24] under the *<uses-permission />* tag [88]. The following sections describe how these permissions have been handled and analyzed.

### 5.3.1 Dataset of permissions

Before starting with the analysis it is essential to create a dataset that collects all the existing permissions and information about them. The aim of having this list is to know the installed apps permissions and to classify them according to the domain that they belong to and their level of danger.

For this work, the team considers that the best implementation of the permission dataset is to have a database formed by two tables, one with the permissions information and another with the groups information (see *Figure 55*).

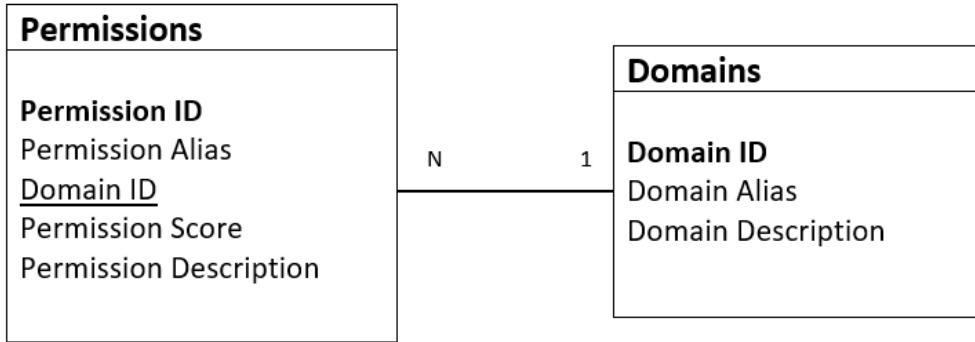


Figure 55: Permissions Dataset Relational Model

The table *Permissions* (*Table 6*) contains all the relevant information regarding Android permissions. As it concerns the identification of this table, the *Permissions ID* field is the constant value of the permission which is stored in the `<uses-permissions/>` tag of the `AndroidManifest.xml`.

The aim of the score field is to grade the permissions according to how exposed the information is as well as the scope of restricted actions you can perform when the system grants you that permission. The values taken by the scores range from 0 to 6 and they are explained in the 5.3.2 section.

Permission ID	Permission Alias	Domain ID	Permission Score	Permission Description
android.permission.ACCEPT_HANDOVER	ACCEPT_HANDOVER		4	Allows a calling app to continue a call which was started in another app. An e
android.permission.ACCESS_BACKGROUND_LOCATION	ACCESS_BACKGROUND_LOCATION		4	Allows an app to access location in the background. Requesting this permis
android.permission.ACCESS_CHECKIN_PROPERTIES	ACCESS_CHECKIN_PROPERTIES		6	Allows read/write access to the "properties" table in the checkin database, to
android.permission.ACCESS_COARSE_LOCATION	ACCESS_COARSE_LOCATION	android.permission-group.LOCATION	4	Allows an app to access approximate location.
android.permission.ACCESS_FINE_LOCATION	ACCESS_FINE_LOCATION	android.permission-group.LOCATION	4	Allows an app to access precise location.
android.permission.ACCESS_LOCATION_EXTRA_COMMANDS	ACCESS_LOCATION_EXTRA_COMMANDS	android.permission-group.SYSTEM_TOOLS	1	Allows an application to access extra location provider commands.
android.permission.ACCESS_MEDIA_LOCATION	ACCESS_MEDIA_LOCATION		4	Allows an application to access any geographic locations persisted in the use
android.permission.ACCESS_NETWORK_STATE	ACCESS_NETWORK_STATE	android.permission-group.NETWORK	1	Allows applications to access information about networks.
android.permission.ACCESS_NOTIFICATION_POLICY	ACCESS_NOTIFICATION_POLICY		1	Marker permission for applications that wish to access notification policy.
android.permission.ACCESS_WIFI_STATE	ACCESS_WIFI_STATE	android.permission-group.NETWORK	1	Allows applications to access information about Wi-Fi networks.
android.permission.ACCOUNT_MANAGER	ACCOUNT_MANAGER	android.permission-group.ACCOUNTS	6	Allows applications to call into AccountAuthenticators.
android.permission.ACTIVITY_RECOGNITION	ACTIVITY_RECOGNITION	android.permission-group.PERSONAL_INFO	4	Allows an application to recognize physical activity.
com.android.voicemail.permission.ADD_VOICEMAIL	ADD_VOICEMAIL	android.permission-group.VOICEMAIL	4	Allows an application to add voicemails into the system.
android.permission.ANSWER_PHONE_CALLS	ANSWER_PHONE_CALLS		4	Allows the app to answer an incoming phone call.
android.permission.BATTERY_STATS	BATTERY_STATS	android.permission-group.SYSTEM_TOOLS	3	Allows an application to collect battery statistics
android.permission.BIND_ACCESSIBILITY_SERVICE	BIND_ACCESSIBILITY_SERVICE		2	Must be required by an AccessibilityService, to ensure that only the system c
android.permission.BIND_APPWIDGET	BIND_APPWIDGET	android.permission-group.PERSONAL_INFO	6	Allows an application to tell the AppWidget service which application can acc
android.permission.BIND_AUTOFILL_SERVICE	BIND_AUTOFILL_SERVICE		2	Must be required by an AutofillService, to ensure that only the system can bind
android.permission.BIND_CALL_REDIRECTION_SERVICE	BIND_CALL_REDIRECTION_SERVICE		5	Must be required by a CallRedirectionService, to ensure that only the system.
android.permission.BIND_CARRIER_MESSAGING_CLIENT_SERVICE	BIND_CARRIER_MESSAGING_CLIENT_SERVICE		2	A subclass of CarrierMessagingClientService must be protected with this perm
android.permission.BIND_CARRIER_MESSAGING_SERVICE	BIND_CARRIER_MESSAGING_SERVICE		5	This constant was deprecated in API level 23. BIND_CARRIER_SERVICES is
android.permission.BIND_CARRIER_SERVICES	BIND_CARRIER_SERVICES		3	The system process that is allowed to bind to services in carrier apps will have
android.permission.BIND_CHOOSER_TARGET_SERVICE	BIND_CHOOSER_TARGET_SERVICE		5	This constant was deprecated in API level 30.

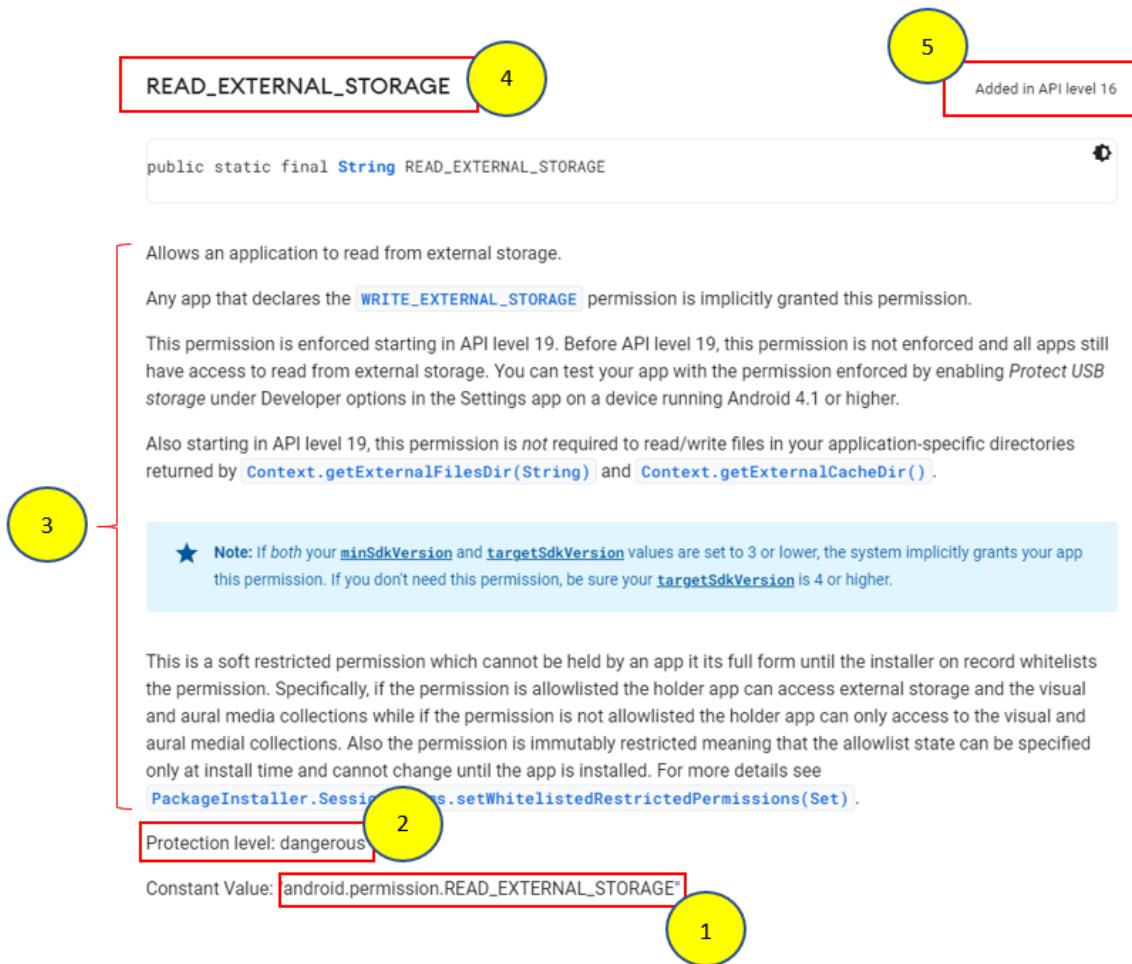
Table 6: Fragment of the table Permissions

The table *Domains* (*Table 7*) contains all the relevant information regarding the different groups which the permissions belong to. This table just stores the identification of the group and an alias and a description to facilitate the understanding of the domain to the user and show him at a high level what functionalities of the device are used by the app.

Domain ID	Domain Alias	Domain Description
android.permission-group.STORAGE	Storage	Access the SD card.
android.permission-group.APP_INFO	Your applications information	Ability to affect behavior of other applications on your device.
android.permission-group.LOCATION	Your location	Monitor your physical location.
android.permission-group.SYSTEM_TOOLS	System tools	Lower-level access and control of the system.
android.permission-group.NETWORK	Network communication	Access various network features.
android.permission-group.ACCOUNTS	Your accounts	Access the available accounts.
android.permission-group.PERSONAL_INFO	Your personal information	Direct access to information about you, stored in on your contact card.
android.permission-group.VOICEMAIL	Voicemail	Direct access to voicemail.
android.permission-group.BLUETOOTH_NETWORK	Bluetooth	Access devices and networks through Bluetooth.
android.permission-group.MESSAGES	Your messages	Read and write your SMS, email, and other messages.
android.permission-group.PHONE_CALLS	Phone calls	Monitor, record, and process phone calls.
android.permission-group.CAMERA	Camera	Direct access to camera for image or video capture.
android.permission-group.DEVELOPMENT_TOOLS	Development tools	Features only needed for app developers.
android.permission-group.AFFECTS_BATTERY	Affects Battery	Use features that can quickly drain battery.
android.permission-group.SCREENLOCK	Lock screen	Ability to affect behavior of the lock screen on your device.
android.permission-group.STATUS_BAR	Status Bar	Change the device status bar settings.
android.permission-group.AUDIO_SETTINGS	Audio Settings	Change audio settings.
android.permission-group.SOCIAL_INFO	Your social information	Direct access to information about your contacts and social connections.
android.permission-group SYNC SETTINGS	Sync Settings	Access to the sync settings.
android.permission-group.MICROPHONE	Microphone	Direct access to the microphone to record audio.
android.permission-group.DEVICE_ALARMS	Alarm	Set the alarm clock.
android.permission-group.WALLPAPER	Wallpaper	Change the device wallpaper settings.
android.permission-group.DISPLAY	Other Application UI	Effect the UI of other applications.
android.permission-group.USER_DICTIONARY	Read User Dictionary	Read words in user dictionary.
android.permission-group.WRITE_USER_DICTIONARY	Write User Dictionary	Add words to the user dictionary.
android.permission-group.BOOKMARKS	Bookmarks and History	Direct access to bookmarks and browser history.
android.permission-group.HARDWARE_CONTROLS	Hardware controls	Direct access to hardware on the handset.
android.permission-group.SYSTEM_CLOCK	Clock	Change the device time or timezone.

Table 7: Fragment of the table Domains

In order to get a permissions dataset as complete as possible, the team relies on the permissions API reference page [79] from the *Android for Developers* official site. This page provides developers with a full list of permissions recognized by Android. In *Figure 56* all the information used in the permissions dataset is highlighted.



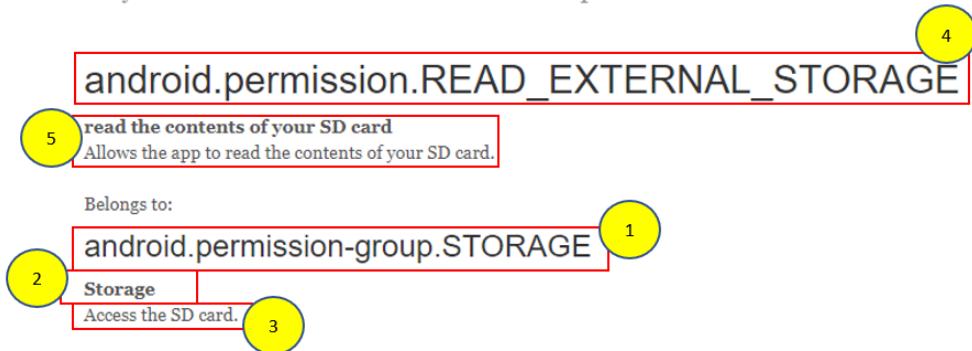
[79] Figure 56: Android Developers, API Reference Page

1 - Permission ID, 2 - Permission Score, 3 - Permission Description, 4 - Permission Alias, 5 - API level (not used)

As additional information, the team has also used the dataset of permissions from the source [androidpermissions.com](http://androidpermissions.com) [80] which completes the list provided by *Android Developers*. Furthermore, this site relates the permissions with their domain and gives an explanation. In *Figure 57* all the information used in the domains dataset is highlighted.

# Android Permissions

All you ever wanted to know about Android permissions



[80] Figure 57: androidpermissions.com

1 - Domain ID, 2 - Domain Alias, 3 - Domain Description, 4 - Permission ID, 5 - Permission Description

## 5.3.2 Classification of permissions

Even though Android already classifies its permissions according to the scope of restricted data and actions that an app can access and perform when the system grants that permission [9], the team decides to adopt a more detailed way of labelling the permissions.

In Table 8 it can be seen this adaptation. The column Score is the value stored in the database that allocates the permission scope and the column Permission Class is the permission classification according to Android.

Score	Permission Class	Description
6	Special	Also known in the analyzer as “Forbidden” by third party apps
5	Deprecated	When a permission constant becomes obsolete in a certain API level, the functionality will be removed in the future and Android asks the developers to not use it or use another constant instead.
4	Runtime	Also known in the analyzer as “Dangerous”, allow the apps to perform restricted actions that more substantially affect the system and other apps.
2	Signature   Privilege Signature	The team decides to consider these two classes as one in the analyzer due to their little difference between each other.
1	Normal	Data and actions present very little risk to the user’s privacy.
0	Unknown	Permissions that are not acknowledged in the dataset of the analyzer

Table 8: Explanatory table of the different levels assigned to the permissions

### 5.3.3 Permissions analysis and scoring

In order to ensure objectivity, the permission analysis is supported by two different *controls*, a quantitative and a qualitative evaluation of the permissions. As a result, each application outputs two different scores that, together, define the access level to restricted data and restricted actions that the app can accomplish, that means, how intrusive an application is. This section of the work goes through all the steps of the permission analysis implemented in the application and explains how both scores are calculated.

A specific section inside the main menu is created in order to perform the permissions analysis. Once the user decides to start the analysis by pressing the button implemented in the fragment, all the logic behind the interface begins the execution. These are the steps that the application follows in order to provide the results of the assessment:

#### 1. Get installed apps

First, the analyzer starts by creating a list of the installed applications. This list is created by the already seen function `getInstalledApps()`. We decided to exclude the system packages from the analysis because it is assumed that they are malware free and do not have any bad or unusual behaviours, so they are not interesting to be analyzed.

#### 2. Information extraction

Once the previous list is already filled with the installed applications, the analyzer proceeds to iterate all the packages and extracts all the relevant information of each app. In order to store this information, the team creates the `AppScore` class.

This class is used to keep the relevant information of the different packages, so an object of this class is created and assigned to each app. The data stored is:

- **PackageInfo:** This object contains the overall information about the contents of a package. This corresponds to all of the information collected from `AndroidManifest.xml` [77].
- **Lists of permissions:** The class contains six different lists of permissions, one for each type of permission. That is, a list for unknown permissions, normal permissions, signature permissions, dangerous permissions, deprecated permissions and forbidden permissions. This is intended to store the permissions according to their classification for a better analysis of the results.
- **HashMap of domains:** This hashmap has the permission groups used by the application as keys and, associated to them as the value, a list of the permissions that belong to the specific group.
- **Category:** This object contains the category flag of the application in a readable way.

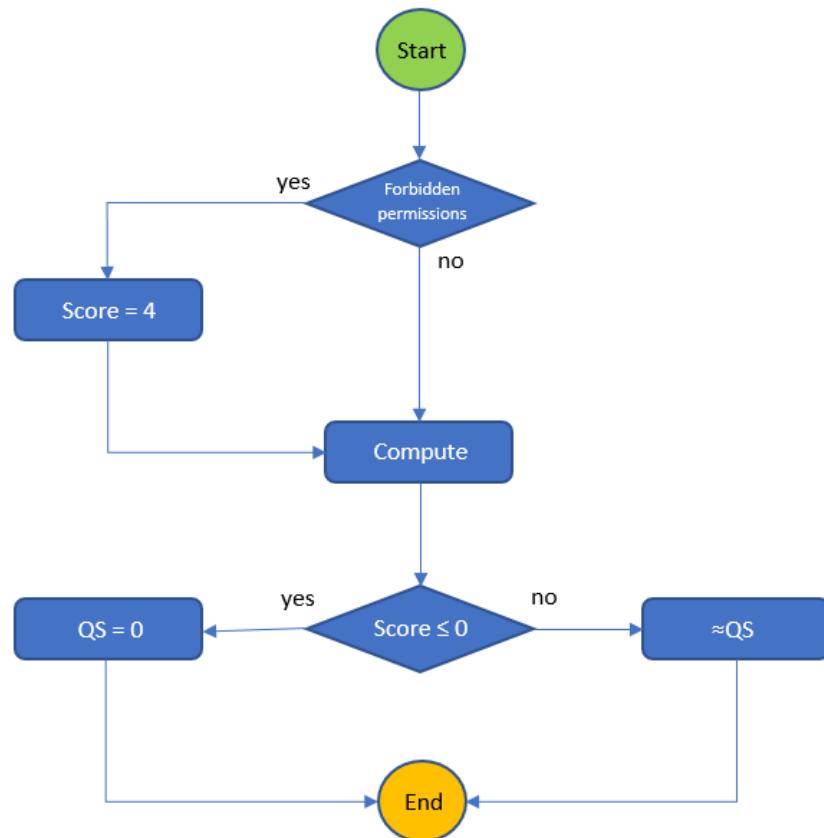
- Quality and Quantity score: These are two scores that rate the application out of 10. They are calculated by two algorithms designed by the team and are described later.

The main information that this analysis needs is the `requestedPermissions[]` field which is a one-dimensional array that contains the `<uses-permissions />` tags, that is, the permissions requested by the application [88]. This field is used to fill the lists of permissions and the hashmap of domains in the `AppScore` class.

In order to correctly arrange the requested permissions, the analyzer accesses the `Permissions` and `Domains` tables of the database and classifies them according to the scope of restricted data and actions that the application can perform and according to the group the permission belongs to.

### 3. Compute Qualitative Score

The Qualitative Score rates each application according to the scope of restricted data that it can access, and the scope of restricted actions that it can perform when the system grants the application those permissions. In order to perform this action, the algorithm that evaluates the application is based on how intrusive it is to the system. *Figure 58* describes how the algorithm works.



*Figure 58: Flowchart of the Qualitative Score algorithm*

1. Start: The algorithm starts by initializing the application score to 1 (10/10 maximum score).
2. If forbidden permissions: Looks if the application requests any forbidden permission.
3. There are forbidden permissions: If the application requests any forbidden permission, the algorithm directly fails the application and resets the maximum score to 0.4 (4/10).
4. Compute QLS: To compute the overall Qualitative Score the algorithm follows this logic, where:

$QLS = \text{Qualitative Score}$

$\text{score} = 1 \text{ or } 0.4$

$\text{forbidden} =$

$\frac{\# \text{ forbidden permissions}}{\# \text{ number of permissions}}$

$\text{dangerous} =$

$\frac{\# \text{ dangerous permissions}}{\# \text{ number of permissions}}$

$\text{deprecated} =$

$\frac{\# \text{ deprecated permissions}}{\# \text{ number of permissions}}$

$\text{unknown} = \frac{\# \text{ unknown permissions}}{\# \text{ number of permissions}}$

$$QLS = (\text{score} - \text{forbidden} - \text{dangerous} - \text{deprecated} - 0.5 * \text{unknown}) \times 10$$

The forbidden, dangerous, deprecated and unknown permissions are those permissions that the user must try to avoid. In order to compute the QLS, the algorithm subtracts these proportions to the maximum score (1) so, in the end, this score approximates to the benign permissions proportion.

5. If the score is negative or equals '0': Check if the Qualitative Score is negative.
6. End: If it is negative, the Qualitative Score is set to 0.
7. End: If it is greater than 0, the Qualitative Score is rounded to 2 decimal places.

#### 4. Compute Quantitative Score

The Quantitative Score rates each application according to the density of requested permissions. In order to perform this action, the algorithm that evaluates the application is based on the size of the field `requestedPermissions[]`.

In 2014 the think tank *Pew Research Center* conducted a study about the Android app permissions where 1,041,336 different applications were analyzed [89]. They concluded that the number of requested permissions was related to the category the application belonged to. That is, depending on the category, the number of permissions tends to be higher or lower. *Figure 59* shows the results of that analysis.

---

**App Permissions Vary a Bit by Category**

<b>Category</b>	<b>Average (mean) # of Permissions</b>	<b>Category</b>	<b>Average (mean) # of Permissions</b>
Communication	9	Education	5
Business	8	Entertainment	5
Casino	7	Family	5
Lifestyle	7	Health & Fitness	5
Role Playing	7	Medical	5
Shopping	7	Music	5
Social	7	Productivity	5
Transportation	7	Racing	5
Travel & Local	7	Simulation	5
Finance	6	Tools	5
Media & Video	6	Trivia	5
Music & Audio	6	Weather	5
News & Magazines	6	Arcade	4
Photography	6	Board	4
Sports	6	Books & Reference	4
Strategy	6	Card	4
Action	5	Casual	4
Adventure	5	Comics	4

Source: Google PlayStore, June 18-Sept 8, 2014.

Note: "Games" was expanded into its subcategories for this list. 8 apps did not have category information.

PEW RESEARCH CENTER

---

*Figure 59: Pew Research average number of permissions by category*

In the application metadata, there exists a field named `category` where it is stored the category flag that indicates the category which it belongs to.

This field accepts 8 valid flags [90]:

- `CATEGORY_GAME`: Category for apps which are primarily games.
- `CATEGORY_AUDIO`: Category for apps which primarily work with audio or music, such as music players.

- CATEGORY\_VIDEO: Category for apps which primarily work with video or movies, such as streaming video apps.
- CATEGORY\_IMAGE: Category for apps which primarily work with images or photos, such as camera or gallery apps.
- CATEGORY\_SOCIAL: Category for apps which are primarily social apps, such as messaging, communication, email, or social network apps.
- CATEGORY\_NEWS: Category for apps which are primarily news apps, such as newspapers, magazines, or sports apps.
- CATEGORY\_MAPS: Category for apps which are primarily maps apps, such as navigation apps.
- CATEGORY\_PRODUCTIVITY: Category for apps which are primarily productivity apps, such as cloud storage or workplace apps.
- CATEGORY\_UNDEFINED: Value when category is undefined.

Merging the results of the *Pew Research Center* analysis with the recognized categories, the outcome of the average number of permissions per category is obtained (see *Table 9*).

<b>Category Flag</b>	<b>Category Value</b>	<b>Average number of permissions</b>
CATEGORY_GAME	0	5
CATEGORY_AUDIO	1	6
CATEGORY_VIDEO	2	6
CATEGORY_IMAGE	3	6
CATEGORY_SOCIAL	4	8
CATEGORY_NEWS	5	6
CATEGORY_MAPS	6	7
CATEGORY_PRODUCTIVITY	7	5
CATEGORY_UNDEFINED	-1	5

*Table 9: Average number of permissions according to the category*

In order to perform the quantitative analysis and to get the Quantitative Score, the algorithm is based on the results from *Figure 60*. It describes how the algorithm works.

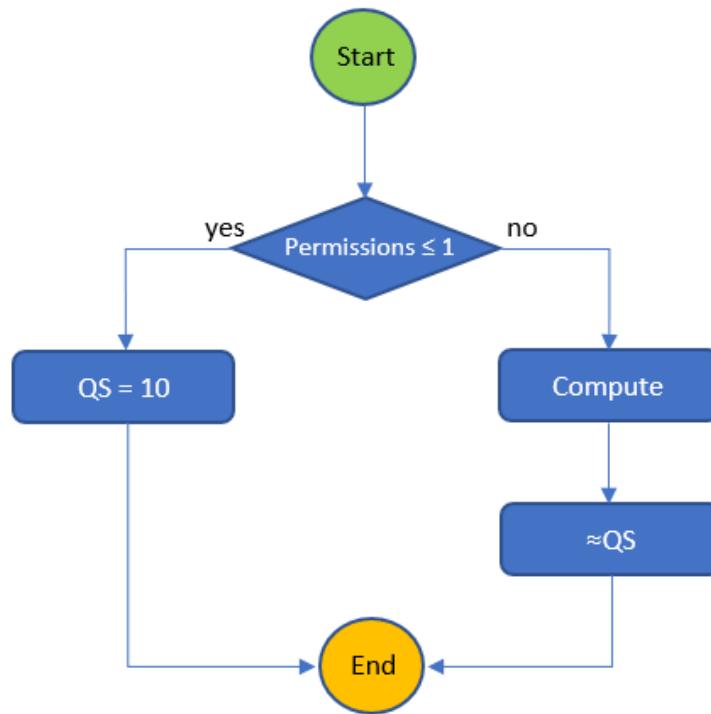


Figure 60: Flowchart of the Quantitative Score algorithm

1. Start: The algorithm starts by initializing the score to 0.
2. Check if the total number of permissions is less or equal to 1.
3. End: If the total number of permissions is less or equal to 1, the algorithm sets the Quantitative Score to 10.
4. Compute QNS: To compute the overall Quantitative Score the algorithm follows this logic, where:

QNS = Quantitative Score

ideal = average number of permissions according to the category

total = total number of permissions requested by the application

$$\text{QNS} = \frac{10 * \text{ideal} - \text{total}}{\text{ideal}}$$

5. End: The Quantitative Score is rounded to 2 decimal places.

This algorithm sets the 10/10 result to those applications whose number of permissions is less than 2, and sets the 0/10 to the applications whose number of

permissions is 10 times the average number of permissions according to the category.

This way, for an application of category *Games* (ideal = 5), depending on the number of permissions, the Quantitative Score is higher or lower:

$$\begin{aligned} \text{QS} & \left\{ \begin{array}{lll} 10 & \text{if} & \text{total} \leq 1 \\ 9 & \text{if} & \text{total} = \text{ideal} \\ 5 & \text{if} & \text{total} = 25 \\ 0 & \text{if} & \text{total} \geq 10 \cdot \text{ideal} \end{array} \right. \end{aligned}$$

## 5. Sort the list of apps

The next and final step of the permission analyzer is to sort into a list the installed apps according to their Final Score. This score is computed as the lowest value between the Quantitative and Qualitative scores.

### 5.3.4 Results

Once the analysis of permissions is done, the results are shown to the user. These results consist on:

- The average of the Final Scores (*Figure 61*).

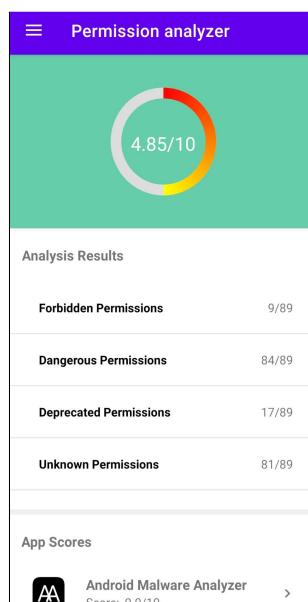


Figure 61: Permission Analysis Results view

- List of the installed applications that request forbidden permissions (*Figure 62* and *Figure 63*).

Permission analyzer		
Analysis Results		
Forbidden Permissions		10/90
 Android Malware Analyzer	Version 1.0	1/6
 Aptoide	Version 9.18.0.0	1/20
 Clean Master	Version 7.4.5	2/81
 Cortar tono de llamada y &lt;br&gt; Cortar música	Version 1	1/14
 FilmoraGo	Version 6.1.0	2/65
 GoAction	Version V1.5.4	2/25
 Logcat Reader	Version 1.7.2	1/4
 Malware Analyzer	Version 1.0	1/10

*Figure 62: Apps that request special permissions*

Permission analyzer			
		Aptoide	
Quantity Score	6.0	Quality Score	0.0
Forbidden Permissions			
dangerous_permissions	android.permission.INSTALL_PACKAGES		
Dangerous Permissions			
Deprecated Permissions			
Signature Permissions			

[91] *Figure 63: Aptoide requests “Install Packages, which is a special permission*

- List of installed applications that request dangerous permissions (*Figure 64* and *65*).

Permission analyzer		
Analysis Results		
Forbidden Permissions		10/90
Dangerous Permissions		85/90
 Adobe Scan	Version 21.05.18-regular	9/17
 Adventures	Version 3.7.35	10/20
 AliExpress	Version 8.29.0	9/35
 Android Malware Analyzer	Version 1.0	2/6
 Aprende Python	Version 2.8.1	8/13
 Aptoide	Version 9.18.0.0	8/20
 Bankia	Version 21.1.6	10/22

*Figure 64: Apps that request dangerous permissions*

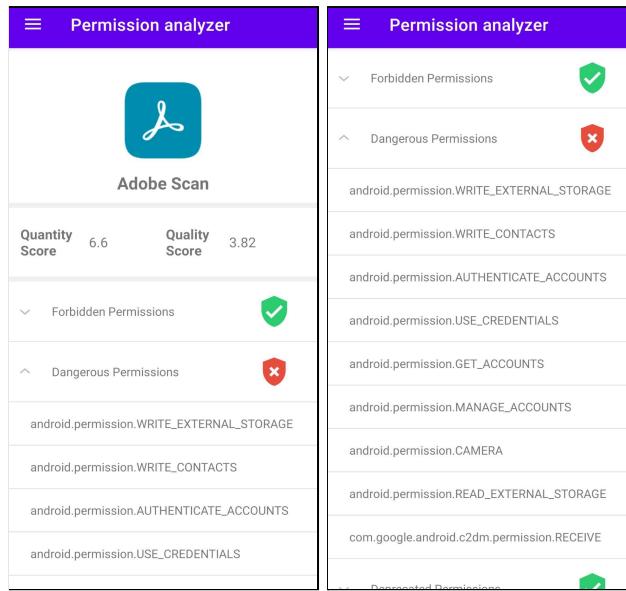


Figure 65: Adobe Scan requests several dangerous permissions

- List of installed applications that request deprecated permissions (Figure 66 and 67).

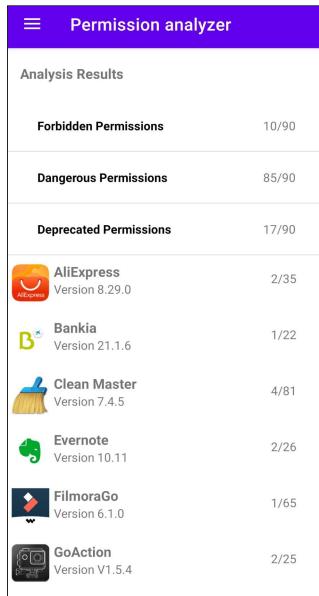


Figure 66: Apps that request deprecated permissions

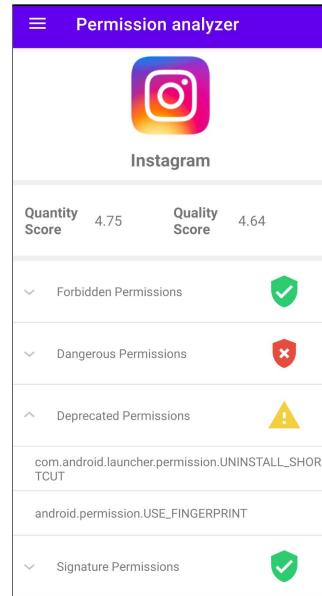


Figure 67: Instagram requests two deprecated permissions:  
“Uninstall\_Shortcut” (see Figure 68) and  
“Use\_Fingerprint” (see Figure 69)

### UNINSTALL\_SHORTCUT

Added in API level 19

```
public static final String UNINSTALL_SHORTCUT
```

! Don't use this permission in your app.  
This permission is no longer supported.

Constant Value: "com.android.launcher.permission.UNINSTALL\_SHORTCUT"

[98] Figure 68: Uninstall Shortcut is a deprecated permission

### USE\_FINGERPRINT

Added in API level 23

Deprecated in API level 28

```
public static final String USE_FINGERPRINT
```

! This constant was deprecated in API level 28.  
Applications should request `USE_BIOMETRIC` instead

Allows an app to use fingerprint hardware.

Protection level: normal

Constant Value: "android.permission.USE\_FINGERPRINT"

[92] Figure 69: Use Fingerprint is a deprecated permission

- List of installed applications that request unknown permissions (*Figure 70 and 71*).

The screenshot shows the 'Analysis Results' section of the Permission analyzer. It lists four categories of permissions: Forbidden Permissions (10/90), Dangerous Permissions (85/90), Deprecated Permissions (17/90), and Unknown Permissions (82/90). Below these are five app entries with their names, versions, and permission counts:

- Adobe Scan: Version 21.05.18-regular, 3/17
- Adventures: Version 3.7.35, 3/20
- AliExpress: Version 8.29.0, 8/35
- Among Us: Version 2021.5.12, 1/4
- Aprende Python: Version 2.8.1, 1/13

Figure 70: Apps that request unknown permissions

The screenshot shows the 'Analysis Results' section for the Instagram app ('Among Us'). It displays two summary scores: Quantity Score (9.2) and Quality Score (8.75). Below this, it lists the following permission types with their status:

- Forbidden Permissions: ✓ (green checkmark)
- Dangerous Permissions: ✓ (green checkmark)
- Deprecated Permissions: ✓ (green checkmark)
- Signature Permissions: ✓ (green checkmark)
- Normal Permissions: ⓘ (blue info icon)
- Unknown Permissions: ⓘ (blue question mark icon)

At the bottom, it shows the specific permission requested: com.android.vending.BILLING.

Figure 71: Instagram requests an unknown permission: "Billing"

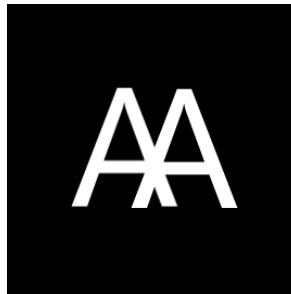
- List of installed applications sorted by Final Score (*Figure 72*).

Permission analyzer		
<b>App Scores</b>		
 <b>Android Malware Analyzer</b> Score: 0.0/10	 <b>Netcat Client</b> Score: 0.67/10	>
 <b>Aptoide</b> Score: 0.0/10	 <b>Malwarebytes</b> Score: 0.73/10	>
 <b>Clean Master</b> Score: 0.0/10	 <b>H.U. Fundación Jiménez Díaz</b> Score: 0.73/10	>
 <b>Cortar tono de llamada y amp; Cortar música</b> Score: 0.0/10	 <b>WhatsApp</b> Score: 2.5/10	>
 <b>FilmoraGo</b> Score: 0.0/10	 <b>Star Walk 2</b> Score: 2.8/10	>
 <b>GoAction</b> Score: 0.0/10	 <b>AliExpress</b> Score: 3.0/10	>
 <b>Logcat Reader</b> Score: 0.0/10	 <b>Skype</b> Score: 3.38/10	>
 <b>Outlook</b> Score: 0.0/10	 <b>Moodle</b> Score: 3.4/10	>
 <b>LinkedIn</b> Score: 0.2/10	 <b>Aprende Python</b> Score: 3.46/10	>
 <b>Netcat Client</b> Score: 0.67/10	 <b>Adobe Scan</b> Score: 3.82/10	>
	 <b>Documentos</b> Score: 4.2/10	>
Permission analyzer		
	 <b>SoloLearn</b> Score: 4.2/10	>
	 <b>Adventures</b> Score: 4.25/10	>
	 <b>Burger King®</b> Score: 4.38/10	>
	 <b>Snapchat</b> Score: 4.38/10	>
	 <b>Spotify</b> Score: 4.4/10	>
	 <b>eduuroamCAT</b> Score: 4.44/10	>
	 <b>Bankia</b> Score: 4.55/10	>
	 <b>Mi Lowi</b> Score: 4.55/10	>
	 <b>Real Guitar</b> Score: 4.6/10	>
	 <b>Instagram</b> Score: 4.64/10	>
	 <b>ElGenero</b> Score: 4.67/10	>
Permission analyzer		
	 <b>GuitarTuna</b> Score: 7.14/10	>
	 <b>MusicAll</b> Score: 7.22/10	>
	 <b>Omnia</b> Score: 7.27/10	>
	 <b>TimeTune</b> Score: 7.27/10	>
	 <b>Geometry Dash Meltdown</b> Score: 7.5/10	>
	 <b>Most Likely</b> Score: 7.5/10	>
	 <b>Hi-Fi Cast</b> Score: 7.73/10	>
	 <b>DreamLab</b> Score: 8.0/10	>
	 <b>Among Us</b> Score: 8.75/10	>
	 <b>Timetable</b> Score: 9.2/10	>
	 <b>NetPal</b>	

*Figure 72: Apps sorted by Final Score*

# 6. Android Malware Analyzer App

This chapter shows the structure of the application developed as a result of the work carried out in this project. This chapter provides a detailed explanation of each layout of the application and the interaction of the user with it. Through this chapter we will refer to the application developed as AMA, (Android Malware Analyzer). We have also designed a logo as shown in *Figure 73*.



*Figure 73: Application Logo*

## 6.1 Fragment Menu

The AMA application has been structured through the use of Fragments. Android Fragments represent a reusable portion of the app's UI that defines and manages its own layout and are attached to an Activity. As shown in *Figure 74 and 75*, AMA consists of 8 main fragments, accessible by a fragment menu, which can be revealed by clicking on the top left three bar icon. The main fragments are:

- **Home:** Entry point of the application. Shows some starting information of the analysis performed.
- **Apps Information:** Lists all the applications installed on the device and when one of them is selected displays information about it.
- **Signature Analyzer:** Fragment in charge of carrying out the signature analysis.
- **Permission Analyzer:** Fragment in charge of carrying out the permission analysis.
- **Log Analyzer:** Fragment in charge of carrying out the log analysis.
- **Previous Results:** Displays the results of previously performed analysis.
- **Server Settings:** Handles the IPs used to establish the connection to the server.
- **About Us:** Shows some information about the project and the three analyses.

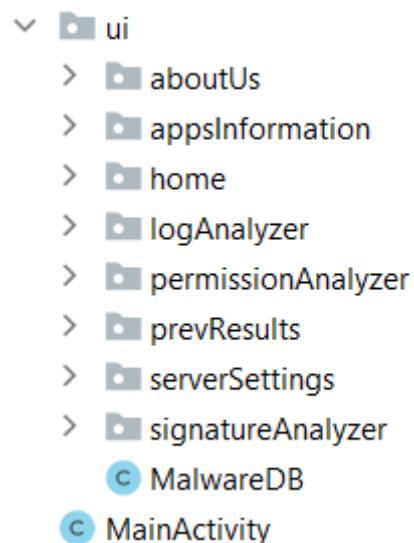


Figure 74: Android project organisation

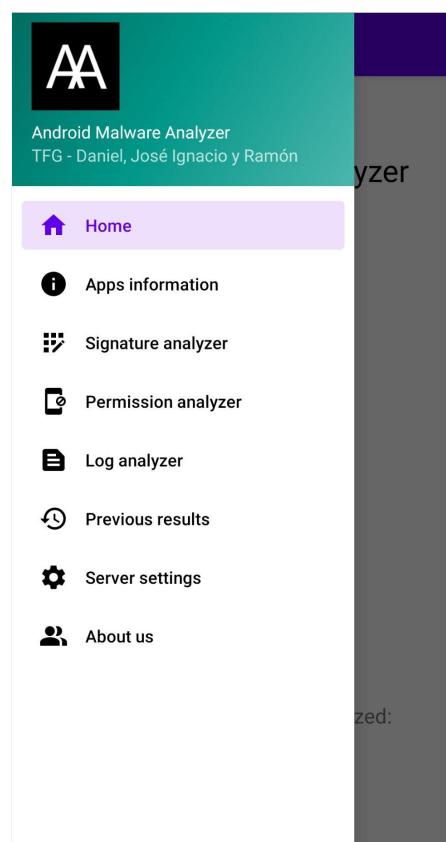
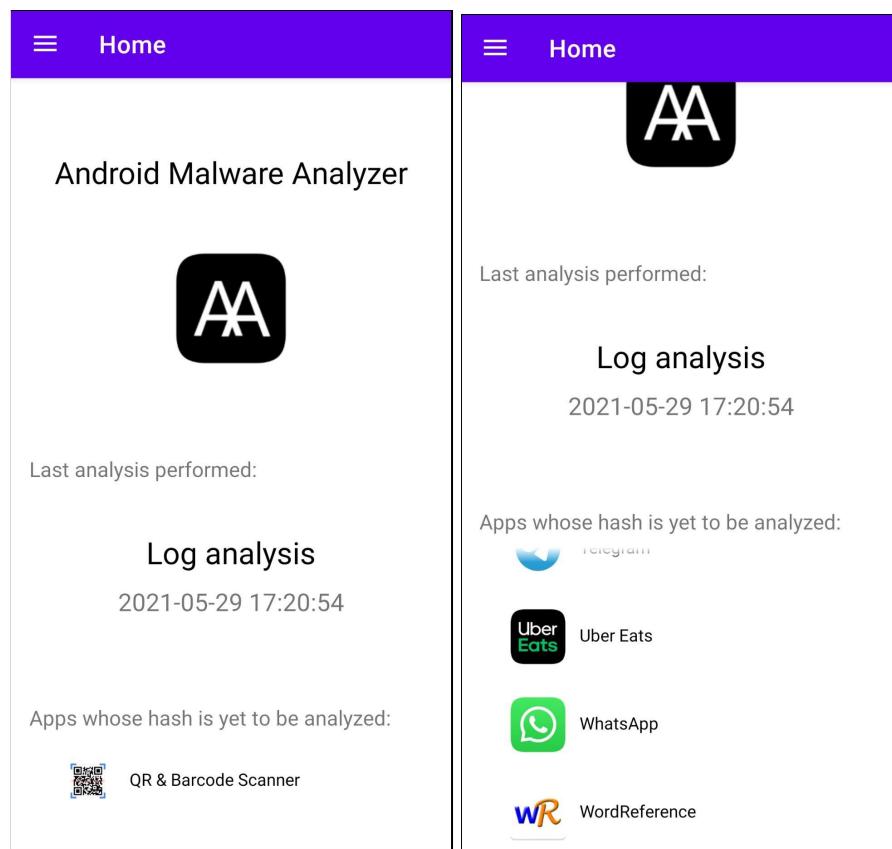


Figure 75: Fragment menu

## 6.2 Home Fragment

The Home Fragment, shown in *Figure 76*, is the visual entry point of the application. It accesses the *PrevResults* database to inform the user about the last analysis performed, displaying the analysis type as well as the date and time. If no analysis is performed, the text “None” is displayed. Below this information, the application shows to the user all the apps whose hash has not been analyzed yet. We have implemented this functionality because we have thought it would be highly recommended to encourage the user to analyze all the applications installed on his device. If all the apps had been analyzed, the text “None” is displayed.



*Figure 76: Home Fragment*

## 6.3 Apps Information Fragment

The *Apps Information Fragment* (Figure 77) shows the user a list of the installed applications. By clicking on a specific app, the user gets the different information that the package provides, the application scores, the requested permissions and a list of the domains that the app has access to.

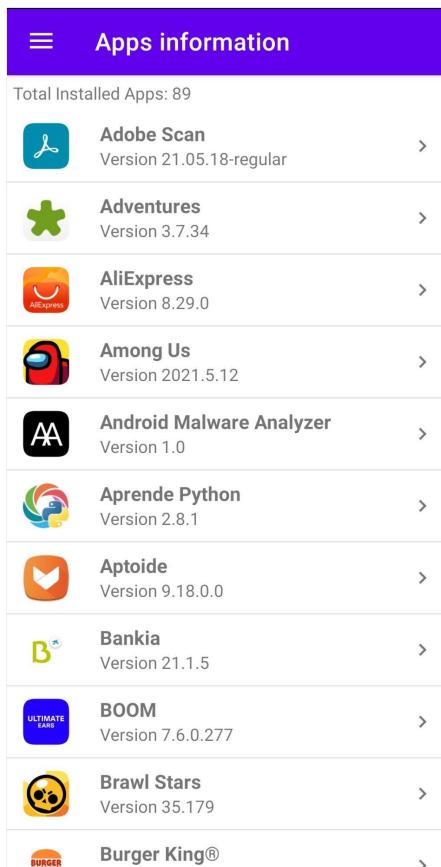


Figure 77: App Information Fragment

The application scores are the quantity and quality results of the assessment performed by the permission analyzer. The overall score of the application is the lowest value of both scores, *Adobe Scan* would have an overall score of 3.82 whereas *Among Us* would have an 8.75 (Figure 78). As seen previously this is based on the permissions requested by the applications. Other relevant information that the package provides is the package name, the path and the categorization of the app.

Next, the fragment shows all the domains of the app, that is, all the permission groups that the application requests. They are displayed in the form of an expandable list that, when clicked, the permissions that belong to that group appear. It also displays to the user a brief description of that domain (see Figure 79).

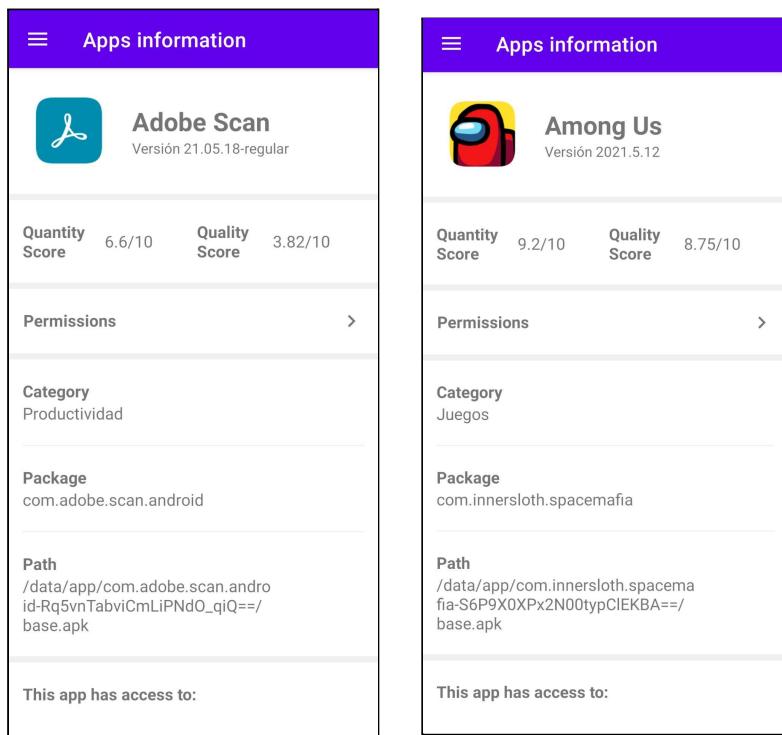


Figure 78: App Details Fragment (Scores and Package Information)

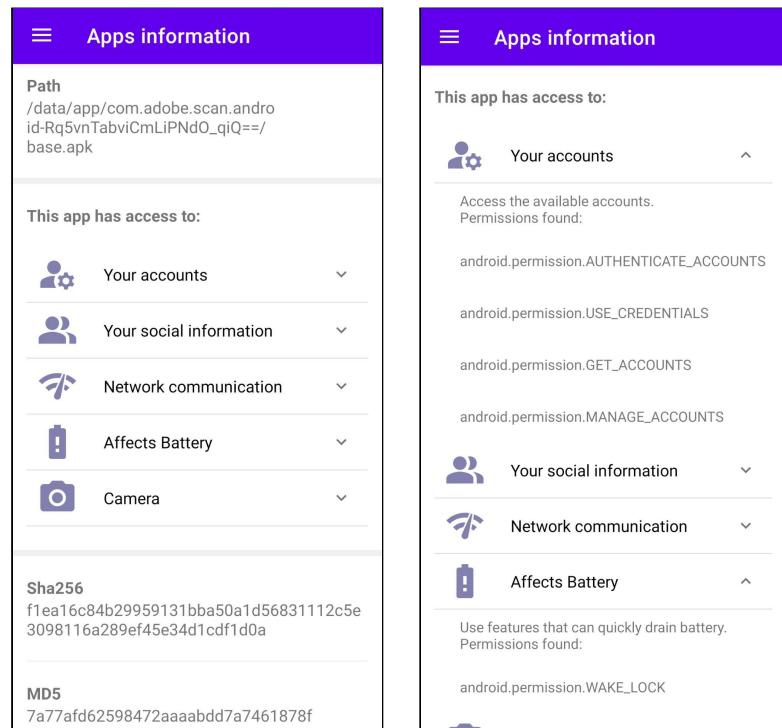
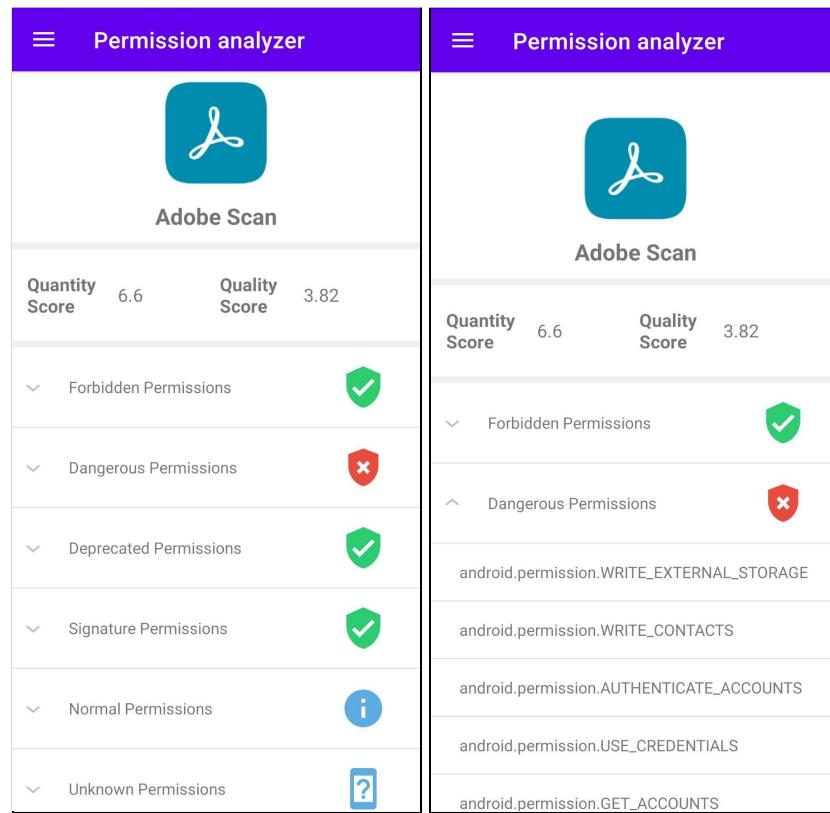


Figure 79: Apps Details Fragment (Domains that Adobe Scan has access to)

If the user clicks on *Permissions* (see *Figure 78*), a new fragment with an expandable list is shown (see *Figure 80*). This list contains the requested permissions organized in classes. Along with the list of permissions, this fragment presents a colour-coded list of icons to indicate the "kindness" of the application according to its permissions. For example, if a green icon appears next to a certain permission class, it means that the application is not requesting any permissions of that class. On the other hand, if any other icon appears it means that the analyzer has found a permission belonging to that group (see *Figure 80*).



*Figure 80: Permissions List Fragment*

Next there are two examples of two different applications, *Timetable* (*Figure 81*) and *FilmoraGo* (*Figure 82*). These applications are chosen as exemplification of a supposedly non-intrusive app and an intrusive app.

The first application (*Figure 81*) is a tool that helps the user to schedule school or college events such as lectures and exams. It just requests four normal permissions so most probably it would not be invasive at all.

On the other hand, the second application is a video editor for Android (*Figure 82*). It requests a total of 65 different permissions where 67.7% are unknown and rare permissions, 17% are forbidden, dangerous or deprecated permissions and 15.3% are normal permissions. According to these proportions, this application could be qualified as very intrusive.

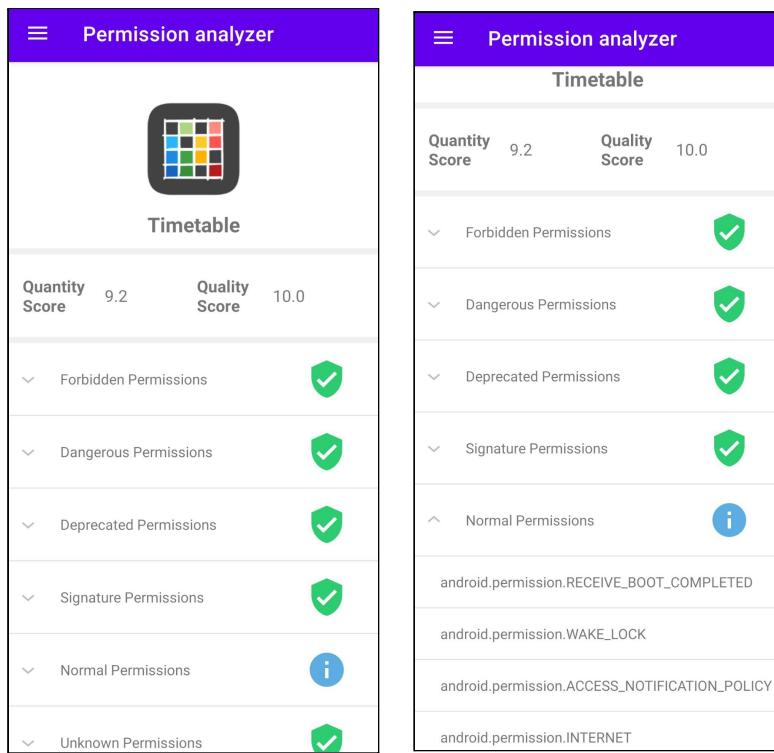


Figure 81: App Details Fragment (Scores and Package Information)

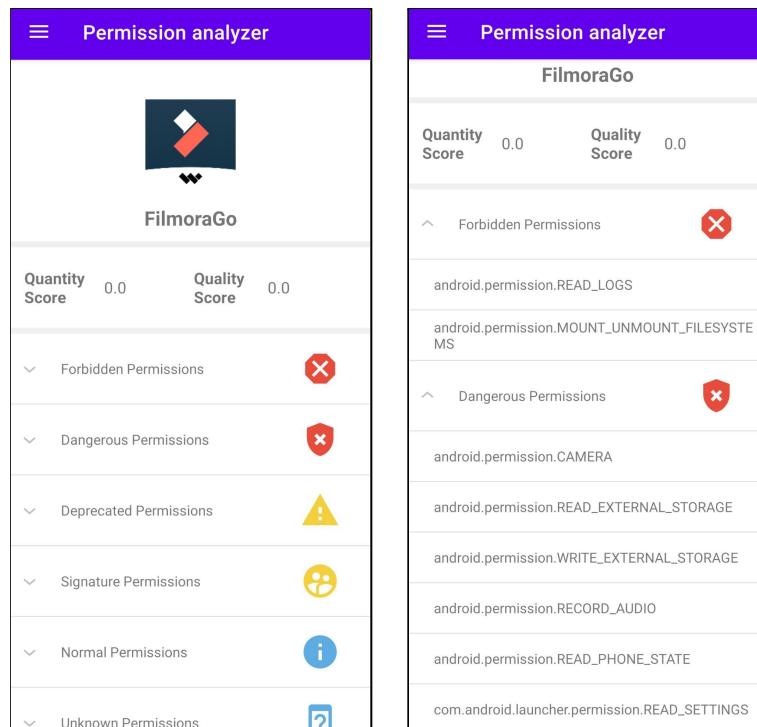


Figure 82: Apps Details Fragment (Domains that Adobe Scan has access to)

By clicking on a permission of the expandable list of *Permissions* (Figure 78) or on any permission inside the domains (Figure 79), the analyzer redirects the user to the *Permissions Fragment* (Figure 83) which is a layout that provides information about that particular permission, such as its classification, the group it belongs to, its description and a brief explanation of what it means to belong to that specific class.

Finally, when the user clicks *Apps with the same permissions* on this very fragment, a list with all the installed applications that own the selected permission are shown (Figure 84).

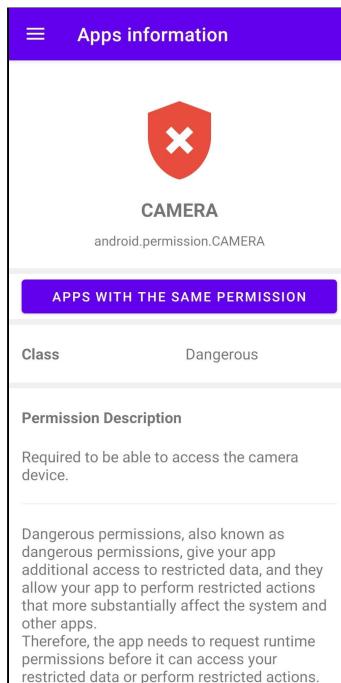


Figure 83: Permission Fragment

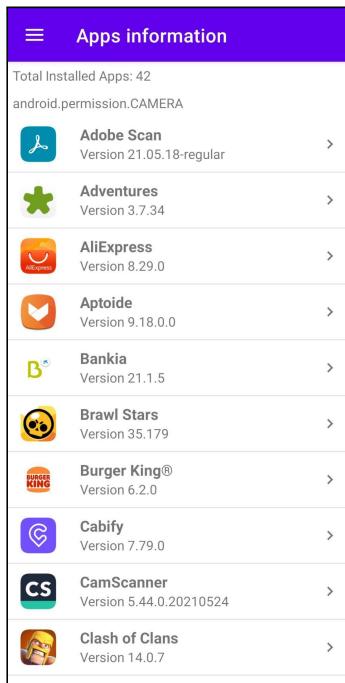
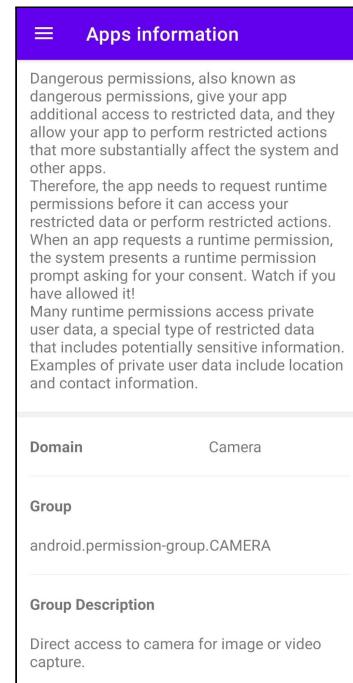


Figure 84: Apps with "Camera" permission

## 6.4 Signature Analyzer Fragment

The graphical part is based on the use of a Recyclerview in charge of displaying the list of applications installed on an Android device within each item. We also included a checkbox in order to select which application the user wants to analyze. It also has a button SELECT ALL to make it easier to select all the existing applications (*Figure 85*).

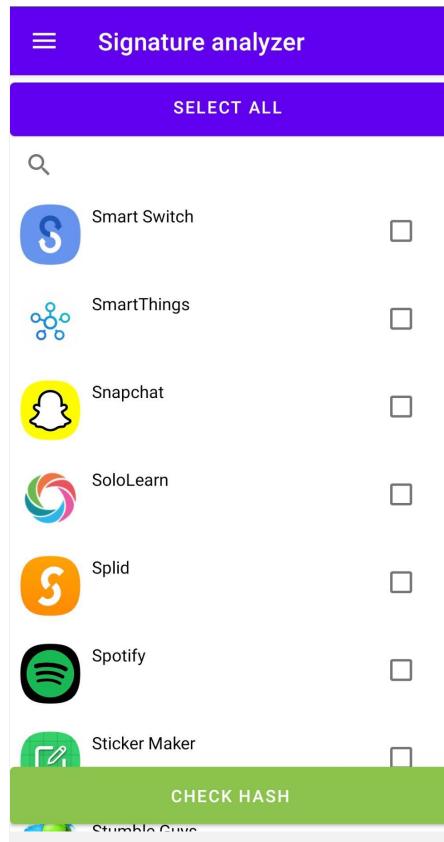


Figure 85: Signature Analyzer Fragment

Due to the fact that a user has a large number of installed and preinstalled applications, it was decided to implement a SearchView in order to provide the user with greater navigability and to make it possible to search for a specific application. This search box can be seen in *Figure 86*.

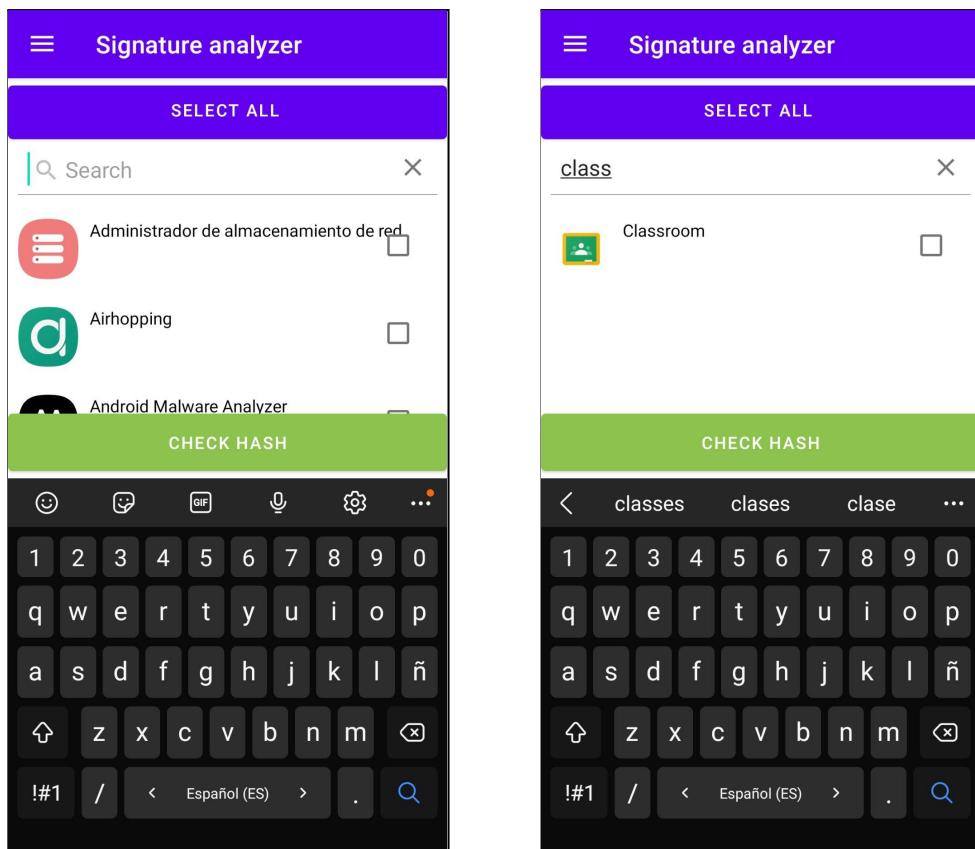


Figure 86: Signature Analyzer Fragment - Search option

The button CHECK HASH, starts the analysis of the applications selected or marked by the checkbox. Once the analysis finishes, a new window appears in which the elements analyzed and the results of the analysis can be seen, as *Figure 87* exhibits. It is to be expected that when performing this analysis on standard apps, which have been downloaded directly from the Google Play Store, the result will always be "no malware found". However, this functionality would be very useful for apps not downloaded through official channels or those obtained from the Google Play Store but which are not "common" apps.

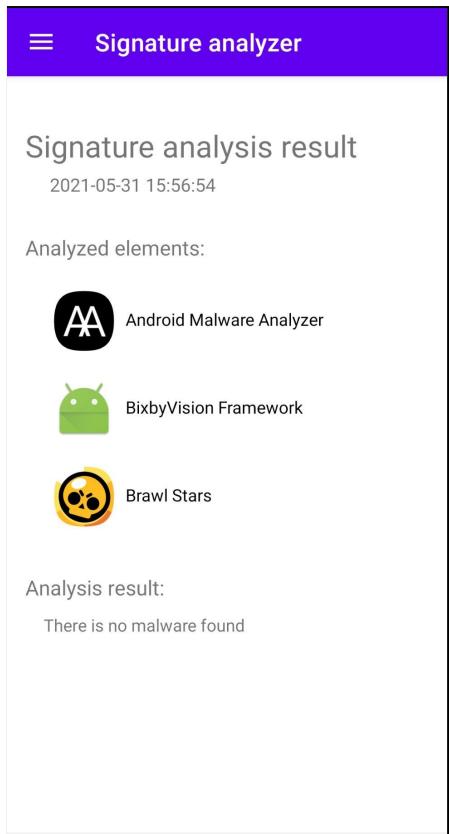


Figure 87: Signature Analyzer Fragment - Search option

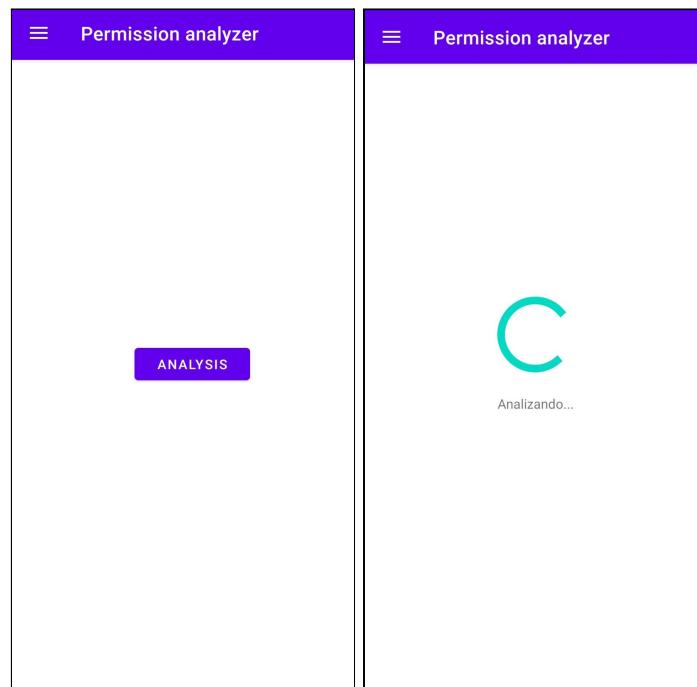
## 6.5 Permission Analyzer Fragment

The *Permission Analyzer Fragment* triggers all the logic behind the permissions analysis. When the user accesses its layout, a button saying “Analysis” is displayed. Once it is pressed, a new thread is created in order to handle just the permissions analysis execution process.

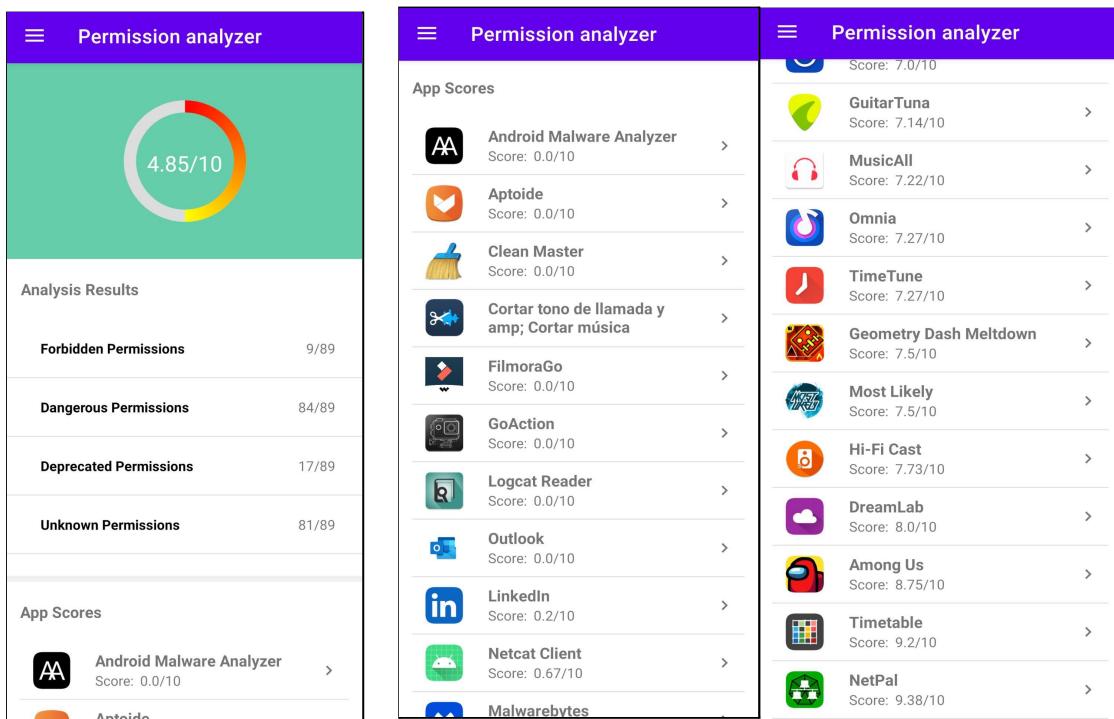
After a few seconds, as explained in Chapter 5, the analysis is completed and the results of the evaluation are shown on screen. In *Figure 88* it is shown the screens related with the beginning of the analyzer, from the user point of view. *Figure 89* shows the result of the analysis: at the top, it is displayed the final score average, calculated as was explained in Chapter 5, of the installed applications. It rates the device vulnerability level against them and also indicates the scope of restricted data that the apps can access, and the scope of restricted actions that apps can perform, when the system grants them permission. The lower the grade, the more vulnerable for the mobile device this app is.

In the same screen, below the final score, an expandable list is shown (*Figure 89*). This list contains the subset of applications that requests at least one forbidden, dangerous, deprecated or unknown permission.

Finally, at the bottom, there is a list that contains the installed applications sorted by final score (*Figure 90*). If the user decides to click in one of the apps, the application is redirected to the *Permission List Fragment* (*Figure 80*).



*Figure 88: Permissions Analyzer Fragment*

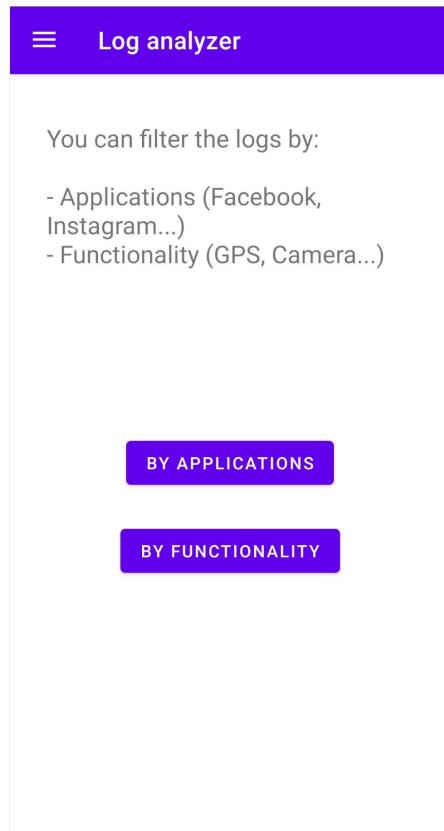


*Figure 89: Permissions Analyzer Fragment*

*Figure 90: Top and Bottom of the applications list sorted by scores*

## 6.6 Log Analyzer Fragment

Firstly, the Log Analyzer Fragment lets the user decide if the apps she/he wants to analyze are user applications (applications installed by him, such as Whatsapp, Instagram, AndroidMalwareAnalyzer, etc) or functionalities (pre-installed apps, such as Camera, Gmail, etc) as shown in *Figure 91*.



*Figure 91: Log Analyzer Fragment*

Depending on the decision the user has made, the list of applications that are displayed changes accordingly, as shown in *Figure 92*. If the connection to the server has been correctly set and selected in the Server Settings Fragment (see section 6.8), an alert message as the one in *Figure 92* is displayed, telling the user to select the apps to monitor.

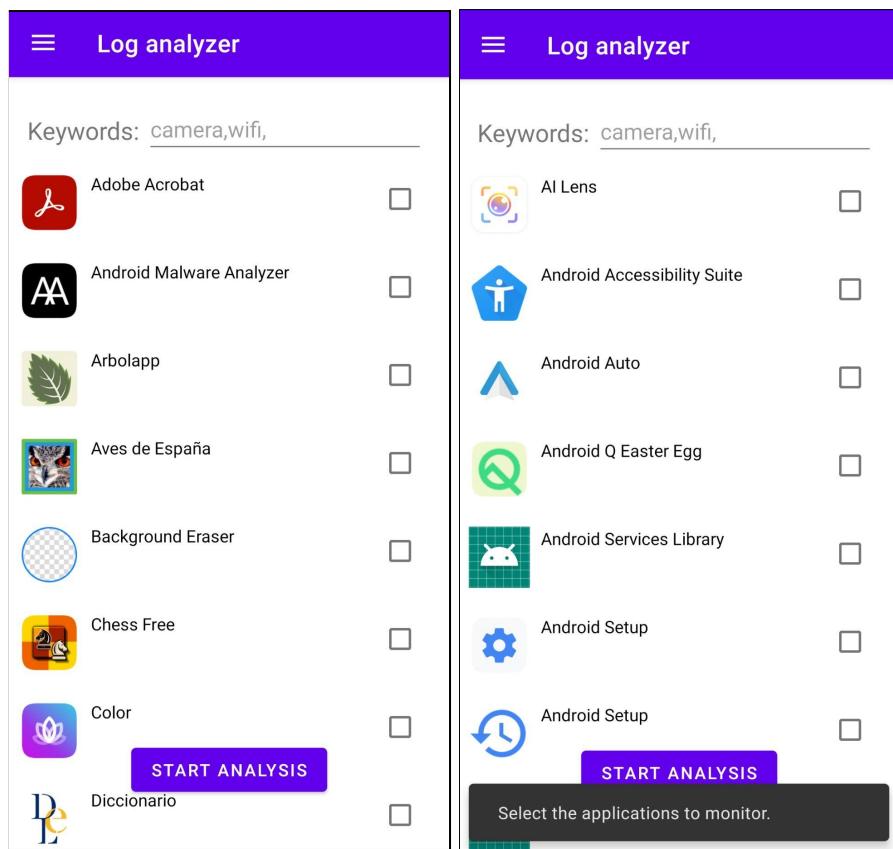


Figure 92: Log Analyzer Fragment - Applications

However, if no address has been correctly set up in the Server Settings Fragment, the alert shows an error message telling the user to add a connection in the Server Settings section (*Figure 93*).

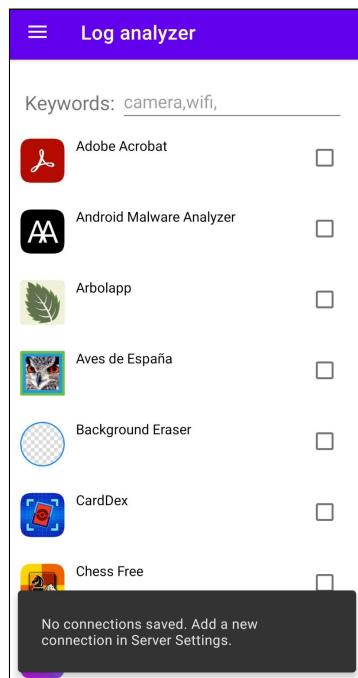


Figure 93: Log Analyzer Fragment - No connections

As *Figure 94* displays, after the previous step the user can select the applications that wants to analyze. If none is selected, all of them are analyzed, which will consume more time compared to selecting some applications. The user can also input some keywords separated by commas, which are used later on during the analysis (see section 5.2.2 for more information). When the user wants to start the analysis, she/he needs to press the *Start Analysis* button. During this time, the connection is established and the application begins to send logs to the server (*Figure 94*).

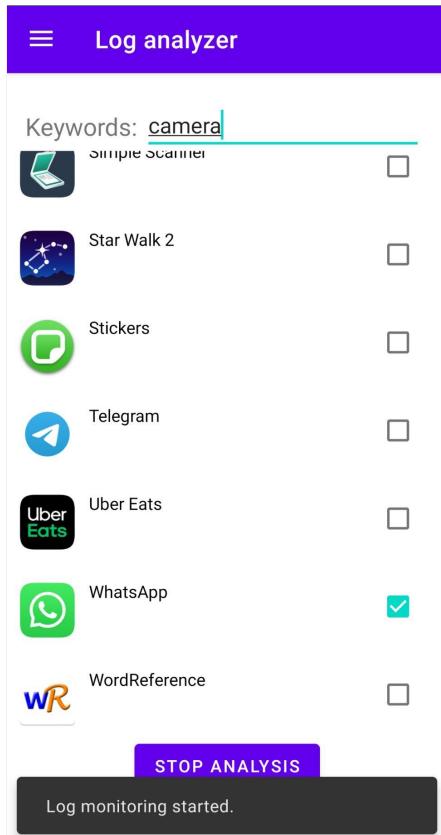


Figure 94: Log Analyzer Fragment - Analysis Started

The user is able to see if something goes wrong through the alert messages that appear on the screen (*Figure 95*). Since the device is now being analyzed, the user can interact with the application she/he wants to analyze, so that AMA gets its logs. As it is a beta version, we recommend accessing the AndroidMalwareAnalyzer once in a while, because it can sometimes get stuck if it is kept in the background for too long.

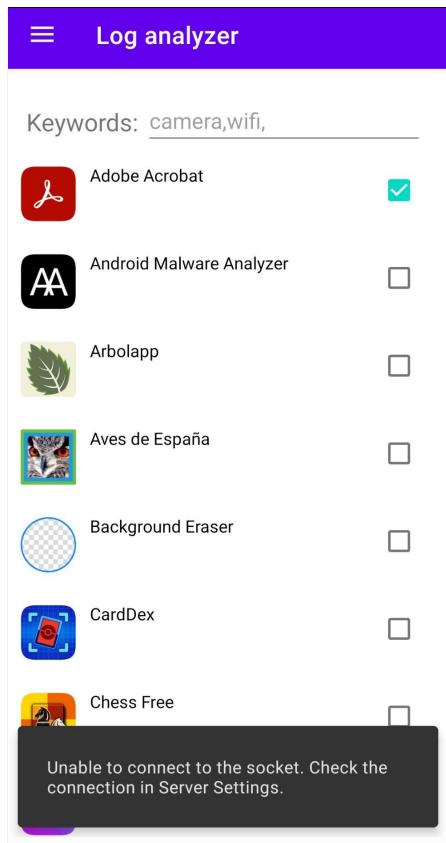


Figure 95: Log Analyzer Fragment - Connection Error

After some time, the user can press the *Stop Analysis* button to stop the connection and retrieve the results. As shown in *Figure 96*, the result first introduces the analyzed elements. Then it shows the apps analyzed, which can be expanded to show the target elements they accessed. Some of them may not be expandable if no target element was recognized. The number displayed indicates how many times that element appeared in a log entry. After that, the keywords analyzed are shown, which can also be expanded to display which application accessed them. Lastly, the user can see if the applications have the permissions needed to access those target elements.

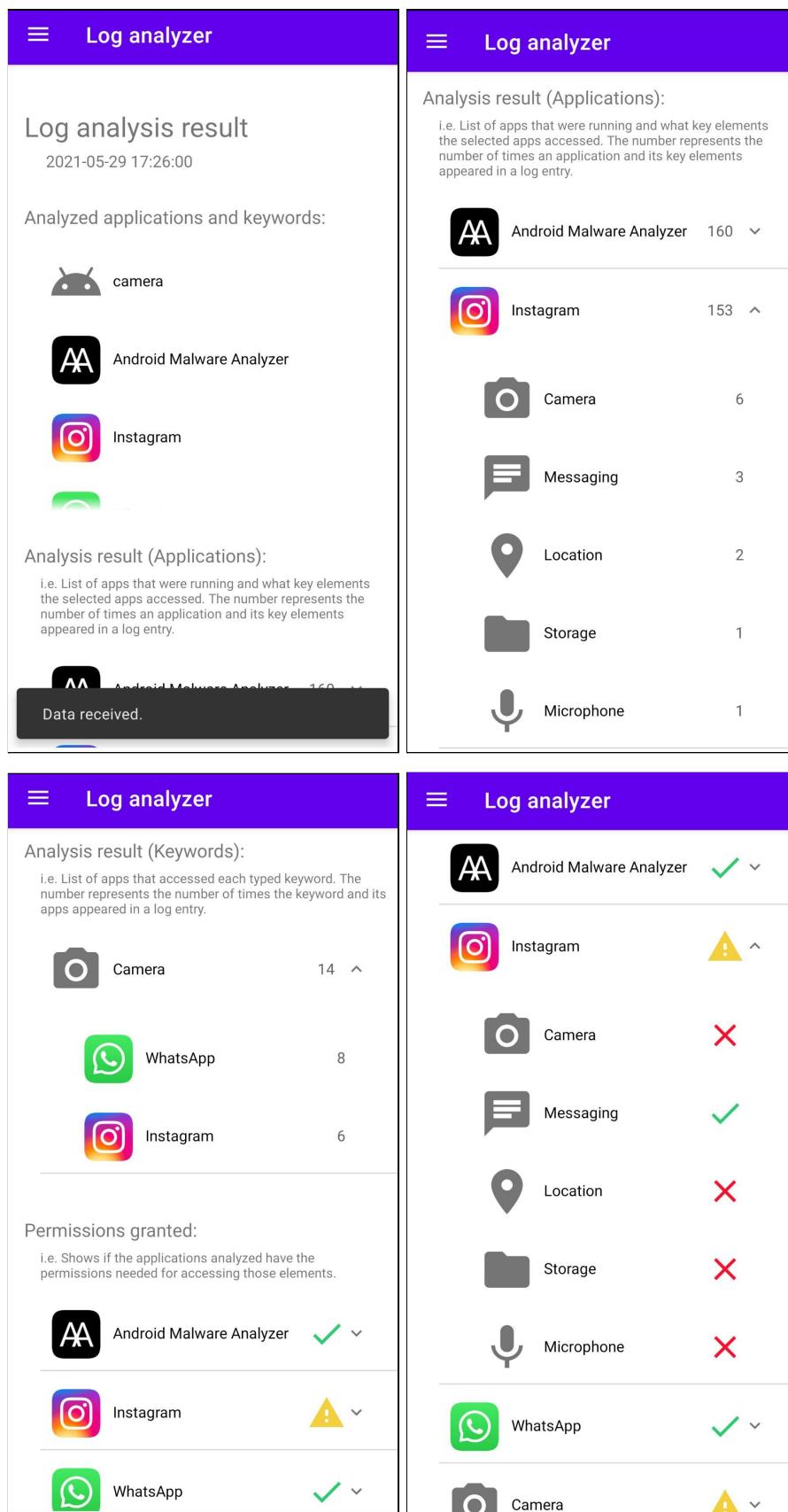


Figure 96: Log Analyzer Fragment - Show Result

## 6.7 Previous Results Fragment

This fragment displays all the log and signature analyses that have been made previously. This is a simple way to keep track of what is happening on your mobile phone with the applications installed. In addition, analysis over time of this data could indicate unwanted operation of installed apps; for example, an app accessing an item too much even though the user is not aware of having that app in use (at least in the foreground). However, we have decided that the permission analysis should not be saved, since it would take up a lot of space due to the amount of information and because it is more interesting to calculate it each time. In section 5.2.1 the contents of the *PrevResults* table are shown (*Table 4*).

As shown in *Figure 97*, each element of the list of analyses shows the type of analysis and the day and time. They are ordered by both the date and time, in descending order.

≡ Previous results	
2021-05-30 11:54:15	Signature analysis result
2021-05-30 11:54:09	Signature analysis result
2021-05-29 17:26:00	Log analysis result
2021-05-29 17:20:54	Log analysis result
2021-05-29 17:18:22	Log analysis result
2021-05-29 17:17:47	Log analysis result
2021-05-29 17:17:16	Log analysis result
2021-05-29 17:15:44	Log analysis result

*Figure 97: Previous Results Fragment*

If one of the results is selected, the specific data is displayed. *Figure 98* displays the results of the signature analysis and log analysis that have been selected.

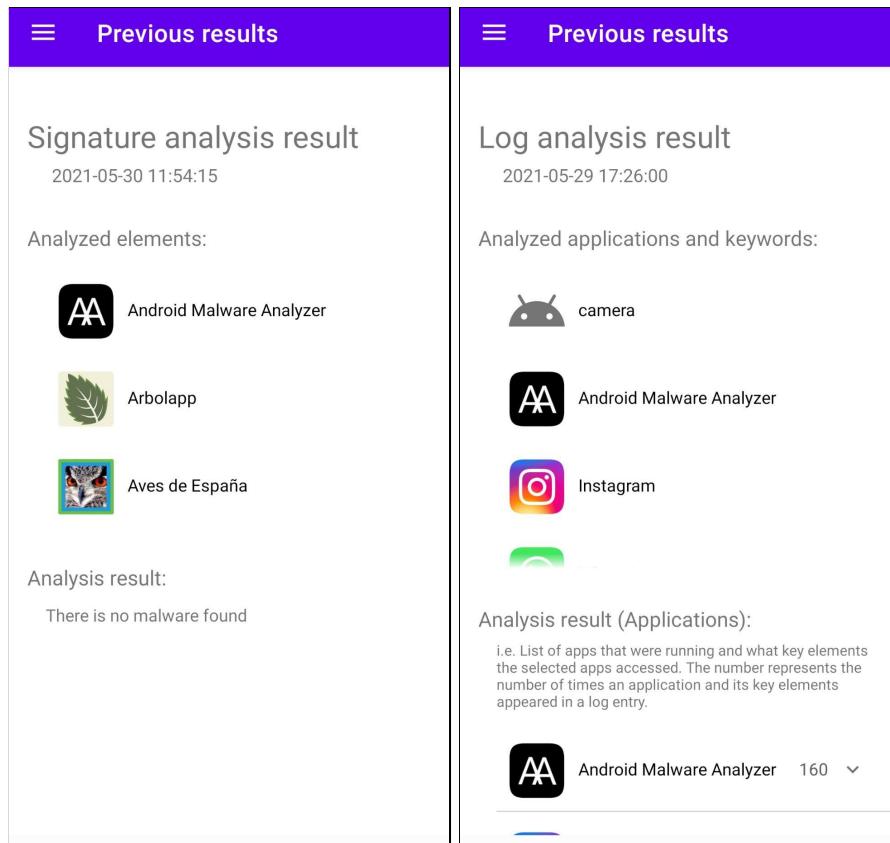


Figure 98: Previous Result Selected

## 6.8 Server Settings Fragment

In this fragment communication with the server is configured. The user can set the IP and Port numbers that are going to be used in the Log Analyzer for connecting to the server. They are stored using SharedPreferences. The user can select each of the saved connections, which is the one used when connecting to the server. As shown in *Figure 99*, there are two buttons: the Add button and the Delete button.

If the Add button is pressed, a popup appears, asking the user to input the IP and Port number in the format IP:Port (*Figure 100*). If the user presses the Cancel button, the operation is discarded.

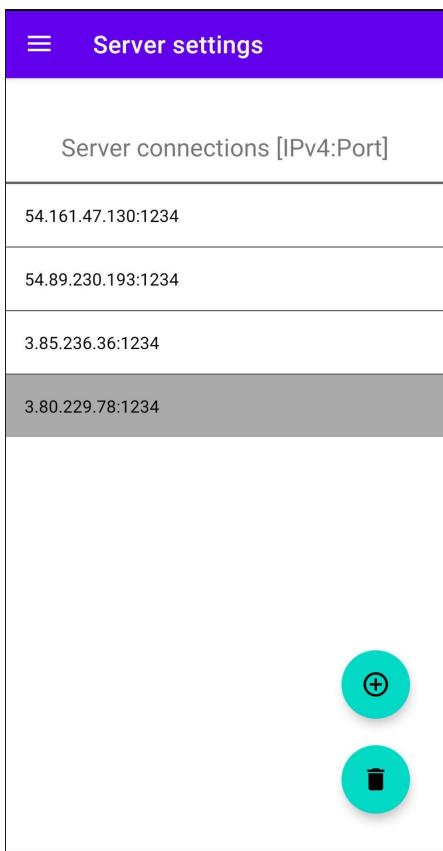


Figure 99: Server Settings Fragment

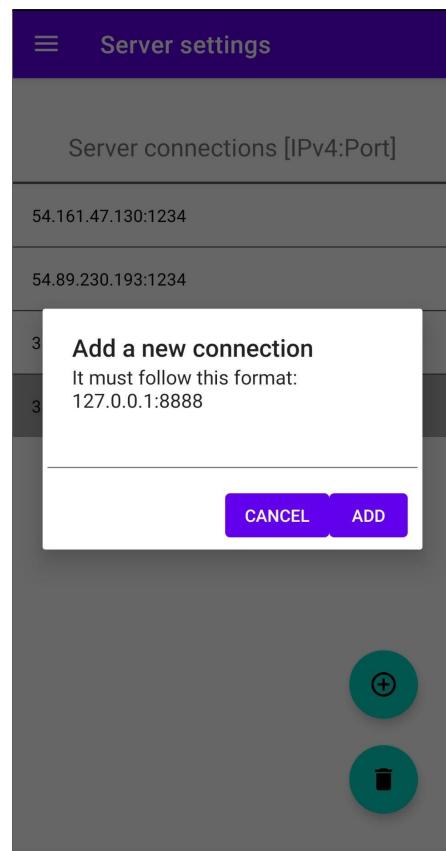


Figure 100: Server Settings Fragment - Add IP:Port

If the user presses the Add button, the user's input is parsed. In case something is wrong with the IP or Port numbers (i.e. if it is not a number, one of the bytes of the IP is inferior to 0 or superior to 255, etc) the connection is not added and an alert message tells the user what went wrong, as shown in *Figure 101*.

If the Delete button is pressed, the selected address is deleted. Once it has been removed, an alert message notifies the user of the deletion (*Figure 102*).

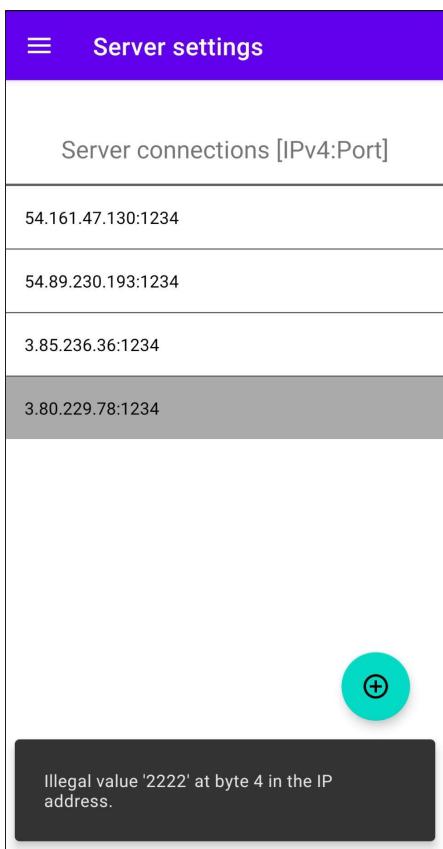


Figure 101: Server Settings Fragment - Wrong Input Alert

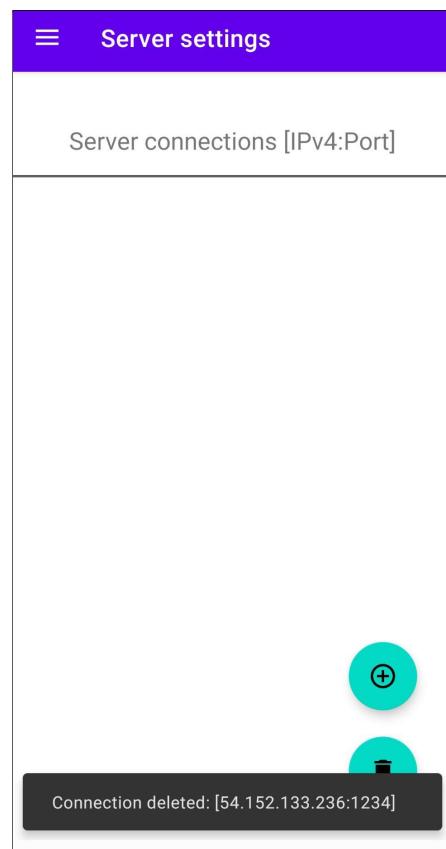
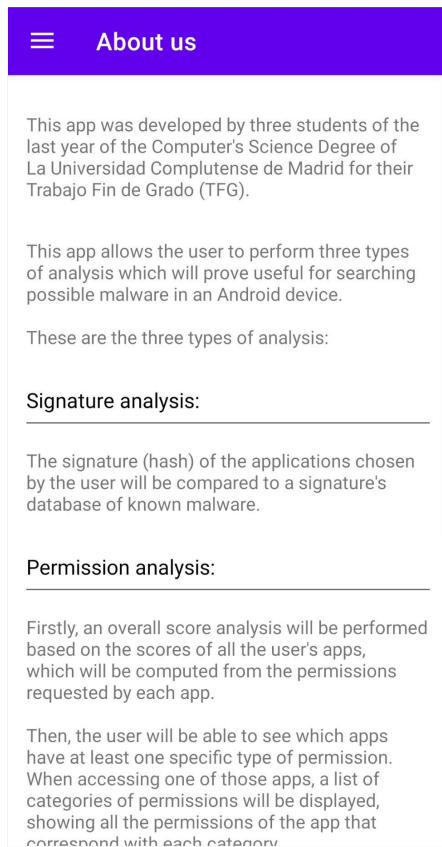


Figure 102: Server Settings Fragment - Delete address

## 6.9 About Us Fragment

This fragment was created to give a bit of context to the users. It shows some information about the project and explains briefly the three different types of analysis. *Figure 103* shows its design.



*Figure 103: About Us Fragment*

# 7. Experimental results

This chapter shows the results of analyzing some applications using the three methods we developed throughout this project. Two types of applications have been used for the analysis: non malicious applications and known malware.

## 7.1 Applications Analyzed

In this section an introduction to the applications we will be analyzing will be provided. Specifically, a brief description, their functionality and the permissions granted/denied will be given. Also, the results obtained from the Apps Information Fragment of AndroidMalwareAnalyzer will be shown and discussed. Information on permissions and signatures will not be addressed, as it will be mentioned in the following sections; instead, we will look at the metadata, as it may provide useful information. An example of this is that, sometimes, the package names can be indicative when detecting malware, as they can be masquerading as a legitimate application but with a different package name or have a package name that indicates the type of malware it is and has no relation to the name of the app itself. Also, the category of the application can offer some insight into the app.

Two types of applications are going to be analyzed: non malicious applications and malware.

As non malicious applications we chose:

- Whatsapp [93] and Instagram [94], as their use is widely spread and they request access to a huge amount of elements.
- TikTok [97], as there is social unrest regarding this application.
- Diccionario de la Lengua Española (DLE) [95], since it will grant a different perspective to the previous ones, as it requests very few permissions and it is a very simple application.

As malware applications we chose:

- Nasip Kismet degilmis [99], since it is a simple Clicker.
- SmartcardService [100], since it is a more complex malware that obtains the phone number and records SMS and phone conversations.

Each of the already mentioned applications will now be introduced in more detail:

- **Instagram**

- Instagram is a social network, owned by Facebook, which is used to share photos, videos, stories, etc. with people all around the world. Each person has a profile (public or private) in which to upload their content to distribute it to their followers, and receive feedback in the form of likes and comments.
- Instagram has a large number amount of functionalities: Posts (which can be photos or videos), Directs (which can be a one-to-one conversion or a group of people; in Directs the user can send photos, videos, text messages,

audios, stickers, emojis...), Stories, Reels, IGTV, Filters, Hashtags, Comments, likes, calls, video calls, etc.

- *Figure 104* shows the permissions required by the app and whether they have been granted or denied. In this case, we decided to deny all of them, which are permissions to the Camera, Contacts, Location, Microphone, Phone and Storage.

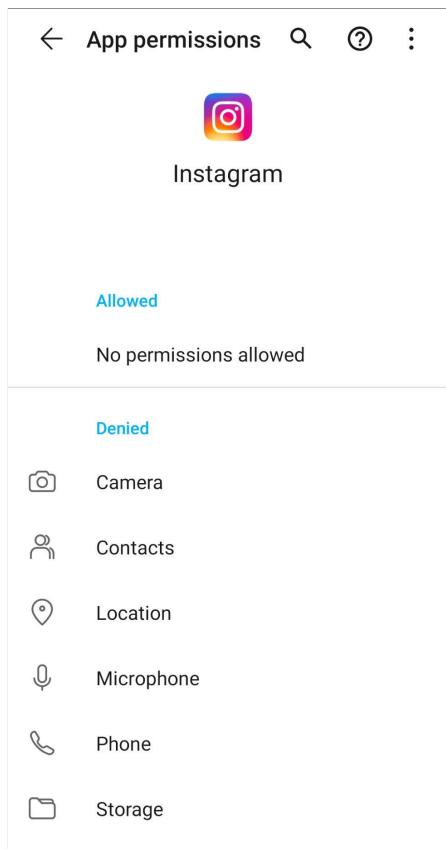


Figure 104: Permission manager - Instagram

- *Figure 105* displays the information obtained from the Apps Information Fragment of AndroidMalwareAnalyzer. It can be seen that the category in which this application falls into is Social & Communication, which is correct for the app's functionalities. Also, the package name of the application is com.instagram.android, which seems to be legitimate.

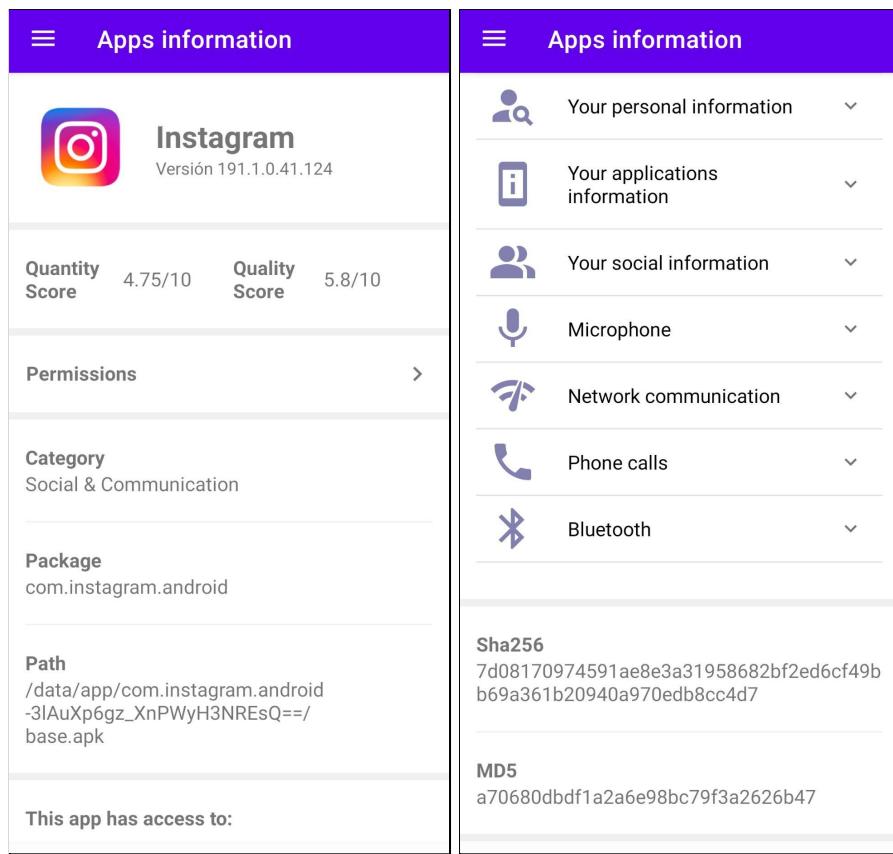


Figure 105: Apps Information - Instagram

#### - WhatsApp

- WhatsApp is a well-known social network, owned by Facebook, where users can send text messages and voice messages, make voice and video calls, and share images, documents, user locations, and other content.
- *Figure 106* shows the permissions required by the app and whether they have been granted or denied. In this case, we decided to grant some of them (Camera, Contacts, Microphone and Storage) and deny the rest (Call logs, Location, Phone, SMS).

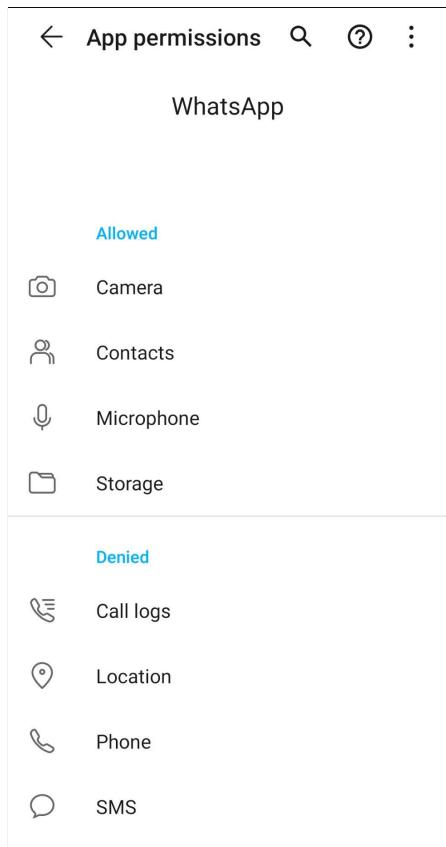


Figure 106: Permission manager - WhatsApp

- *Figure 107* displays the information obtained from the Apps Information Fragment of AndroidMalwareAnalyzer. It can be seen that the category in which this application falls into is Social & Communication, which is correct for the app's functionalities. Also, the package name of the application is com.whatsapp, which seems to be legitimate.

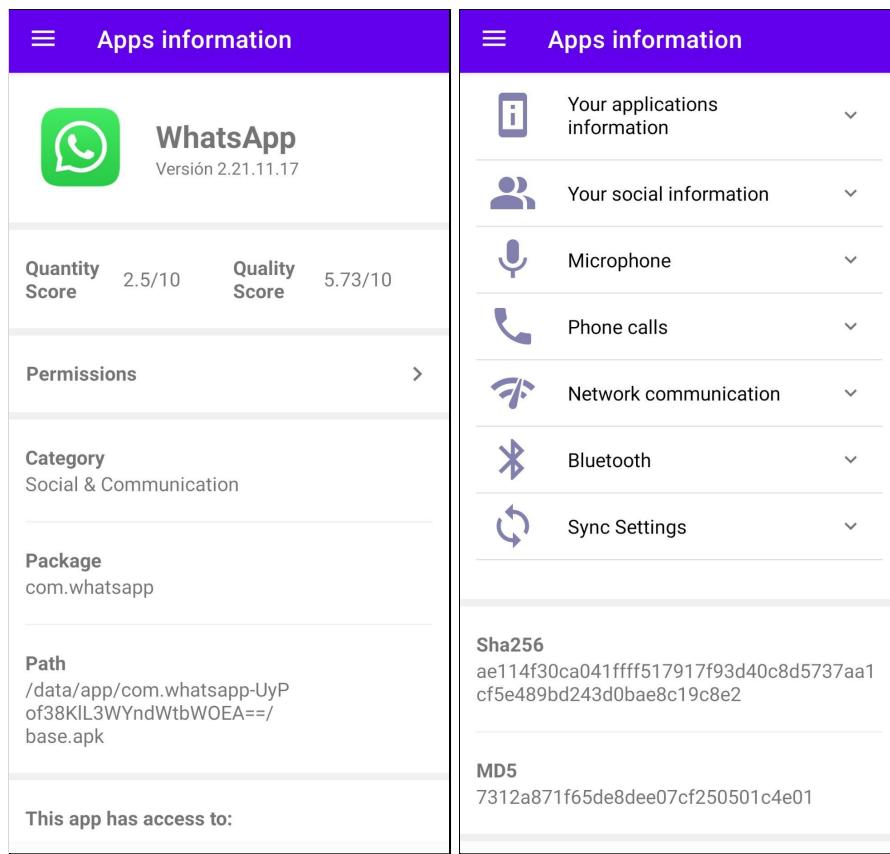


Figure 107: Apps Information - WhatsApp

#### - TikTok

- TikTok, known in China as Douyin, is a Chinese video-sharing focused social networking service owned by Chinese company ByteDance. The social media platform is used to make a variety of short-form videos, from genres like dance, comedy, and education, that have a duration from 15 seconds to one minute.
- *Figure 108* shows the permissions required by the app and whether they have been granted or denied. In this case, we decided to deny all of them, which are permissions to the Storage, Calendar, Camera, Contacts and Microphone.

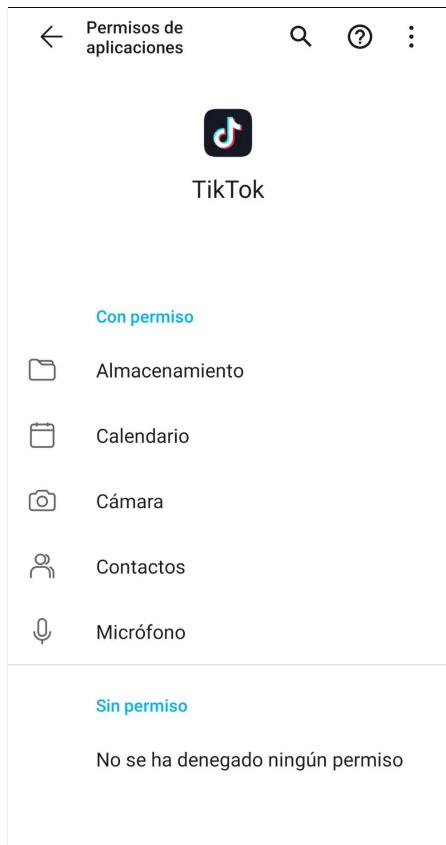
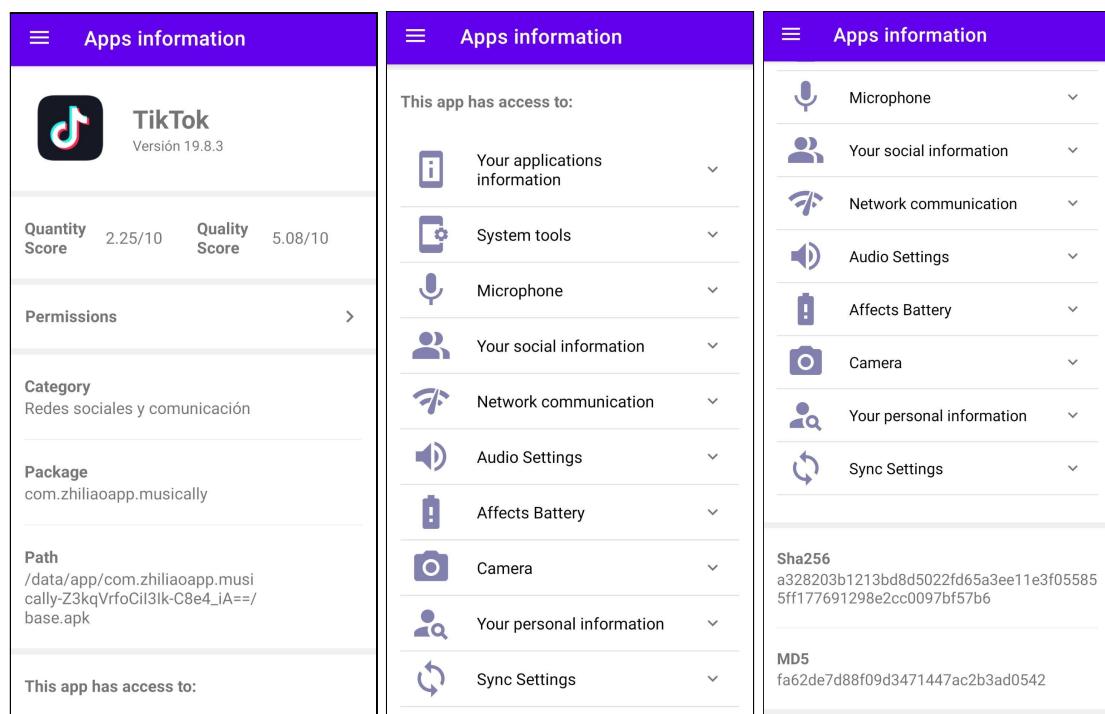


Figure 108: Permission manager - TikTok

- *Figure 109* displays the information obtained from the Apps Information Fragment of AndroidMalwareAnalyzer. It can be seen that the category in which this application falls into is Social & Communication, which is correct for the app's functionalities. The package name of the application is com.zhiliaoapp.musically, which seems to be quite suspicious since the current name is *TikTok*. However, it might be legitimate because *Musically* is the former name of the app.



#### - Diccionario de la Lengua Española (DLE)

- DLE is the official application that the Real Academia Española (RAE) and the Asociación de Academias de la Lengua Española (ASALE) made available to consult the Spanish Dictionary.
- Its main functionality is the search of the meaning of specific words in the dictionary, being able to apply different filters. It also has some links that allow the user to obtain information about the app and the entities behind its development.
- *Figure 110* shows the permissions required by the app and whether they have been granted or denied. In this case, it has not required any.

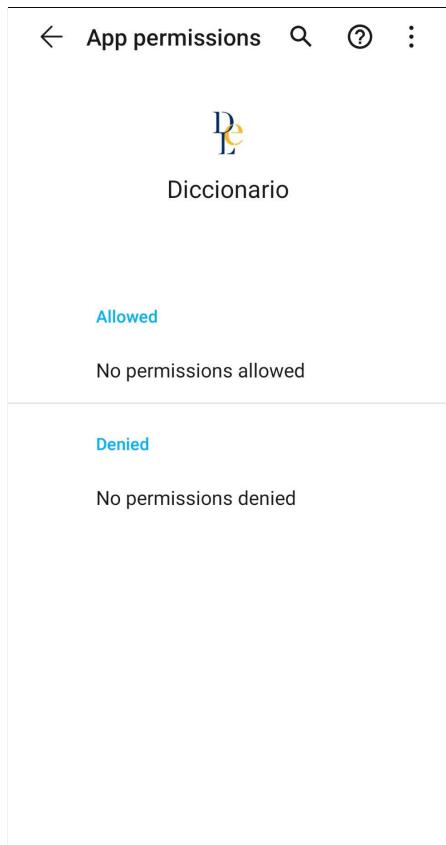


Figure 110: Permission manager - DLE

- *Figure 111* displays the information obtained from the Apps Information Fragment of AndroidMalwareAnalyzer. It can be seen that the category in which this application falls into is Productivity, which is correct for the app's functionalities. Also, the package name of the application is es.rae.dle, which seems to be legitimate.

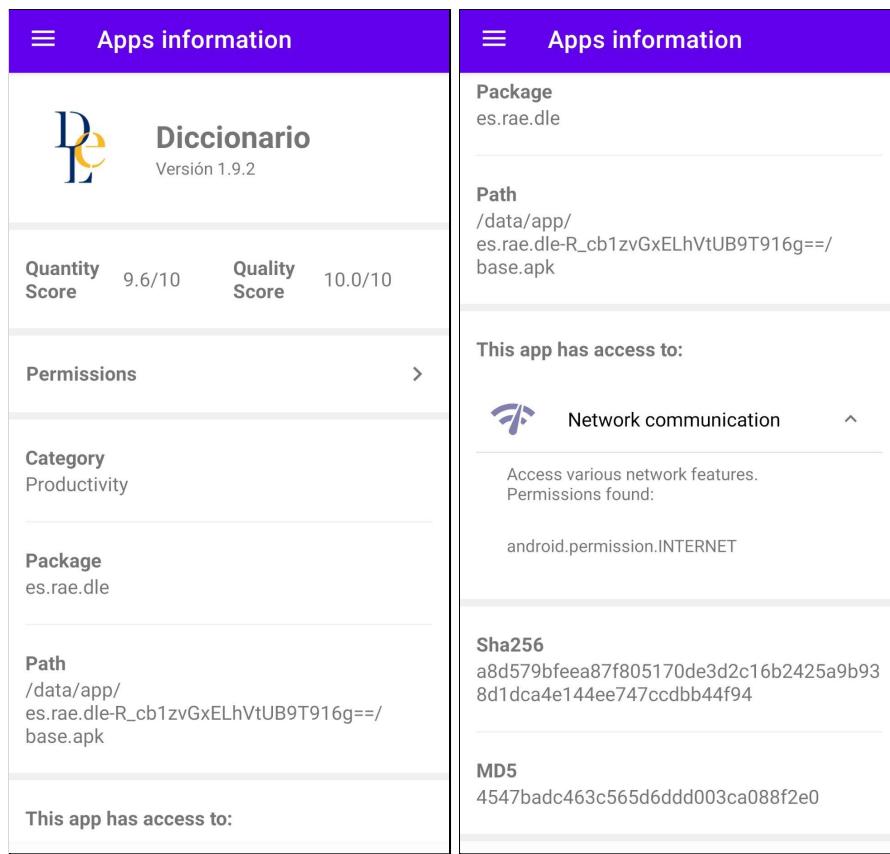


Figure 111: Apps Information - DLE

#### - Nasip Kismet degilmis

- It is a supposed TV remote control that actually does not work as a remote control, but as a Clicker.
- *Figure 112* displays the information obtained from the Apps Information Fragment of AndroidMalwareAnalyzer. It can be seen that the category in which this application falls into is Undefined, which is already suspicious. Also, the package name of the application is com.ndsonkentucky.kuma, which also seems suspicious.

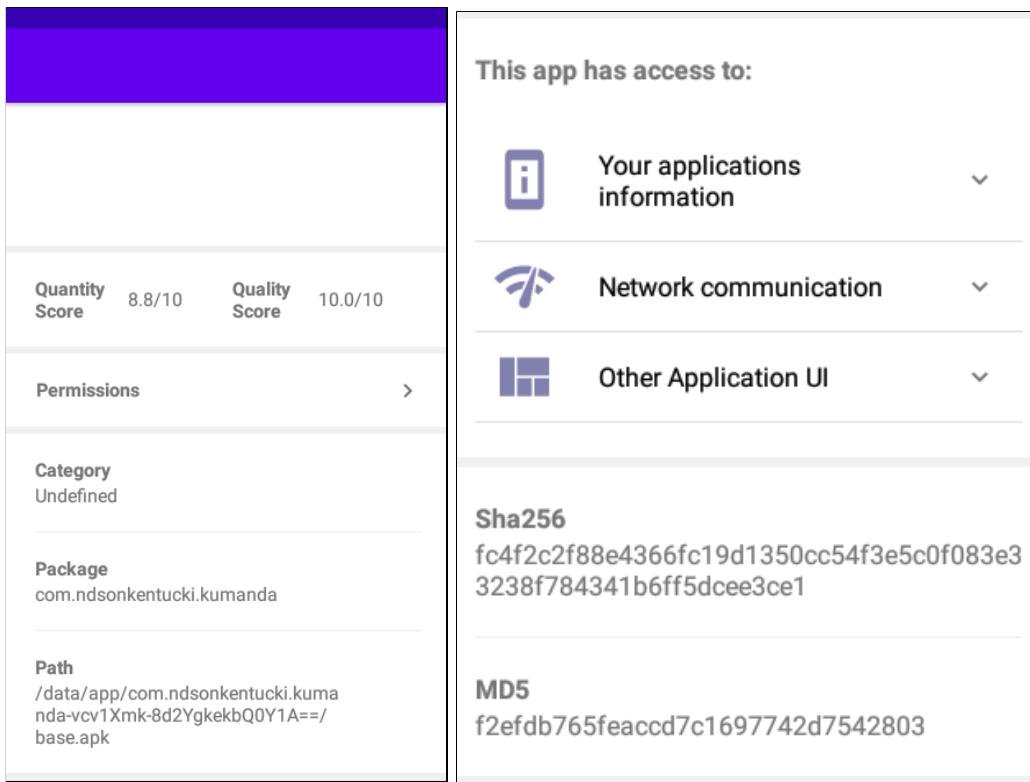


Figure 112: Apps Information - Nasip Kismet degilmis

- **SmartcardService**

- It is a malware that obtains the phone number, MAC address, device usage and records SMS and voice conversations.

## 7.2 Permission Analysis

In this section, the permission analyzer will examine the apps mentioned in the previous section. Based on the result, the permissions requested will be discussed to see if their use is legitimate or if it contradicts its alleged functionality.

- **Instagram**

- *Figure 113* shows the overall score obtained by analyzing the app with the permission analyzer. As seen, it scored a 4.75 out of 10, which is a low score. This is due, as we will later see, to its huge number of permissions and the high ratio of dangerous permissions.

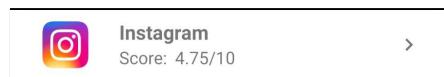


Figure 113: Overall Score - Instagram

- Accessing the result of the analysis, shown in *Figure 114*, it can be seen that the application requests several types of permissions. Specifically, this app requests 15 dangerous permissions, 2 deprecated, 14 normal and 11

unknown, which gives it a rating of 5.8 out of 10 in quality. In terms of quantity, it requests 42 permissions in total, hence it has a 4.75. Of all of them, the most notable permissions are those related to the Camera, Accounts, Contacts, Location, Microphone, Phone, Storage and Billing, which make sense within all the functionalities that the app offers. All in all, it can be appreciated that this application requests a great number of permissions, which could be dangerous in case the application were malicious. In this case, this application is the legitimate one, and those permissions are needed to perform several of the functionalities of the app.

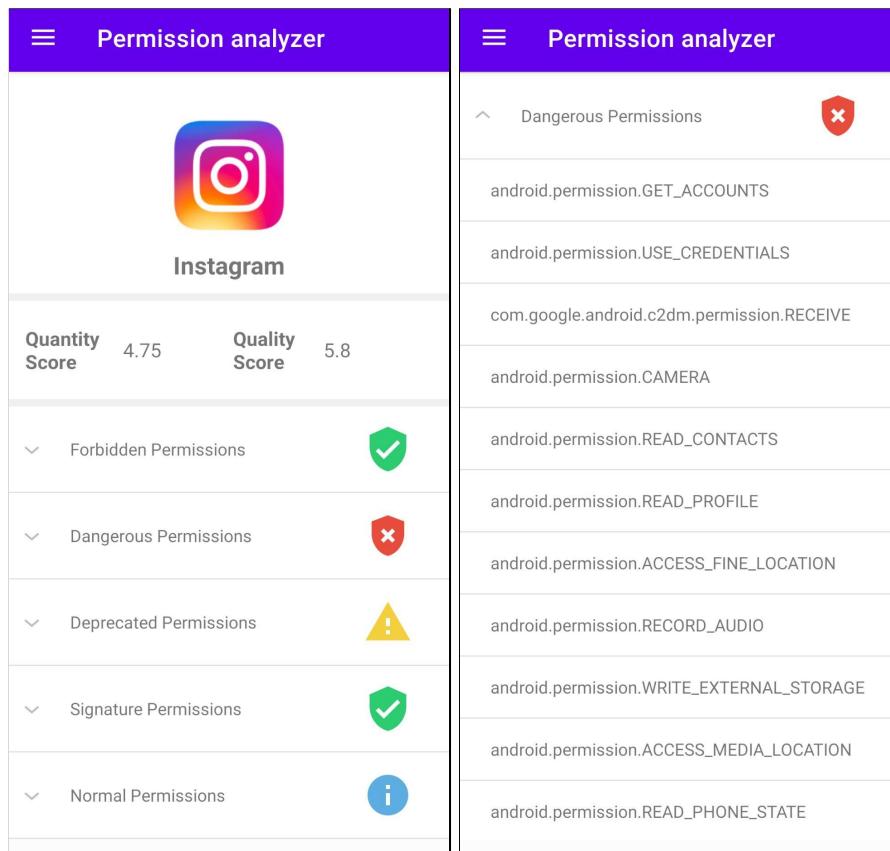


Figure 114: Permission Analyzer - Instagram

## - WhatsApp

- *Figure 115* shows the overall score obtained by analyzing the app with the permission analyzer. As seen, it scored a 2.5 out of 10, which is a very low score. This is due, as we will later see, to its large number of permissions and the poor balance of permission types according to the security levels.

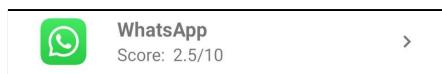


Figure 115: Overall Score - Whatsapp

- Accessing the result of the analysis, shown in *Figure 116*, it can be seen that the application requests several types of permissions. Specifically, this app requests 22 dangerous permissions, 3 deprecated, 1 signature, 20 normal and 14 unknown, which gives it a rating of 5.73 out of 10 in quality. In terms of quantity, it requests 60 permissions in total, hence it has a 2.5. Of all of them, the most notable permissions are those related to the Camera, Accounts, Contacts, Location, Microphone, Phone, SMS, Read Call Log, NFC, Biometric, Storage and Billing, which make sense within all the functionalities that the app offers. All in all, it can be appreciated that this application requests a great number of permissions, which could be dangerous in case the application were malicious. In this case, this application is the legitimate one, and those permissions are needed to perform several of the functionalities of the app.

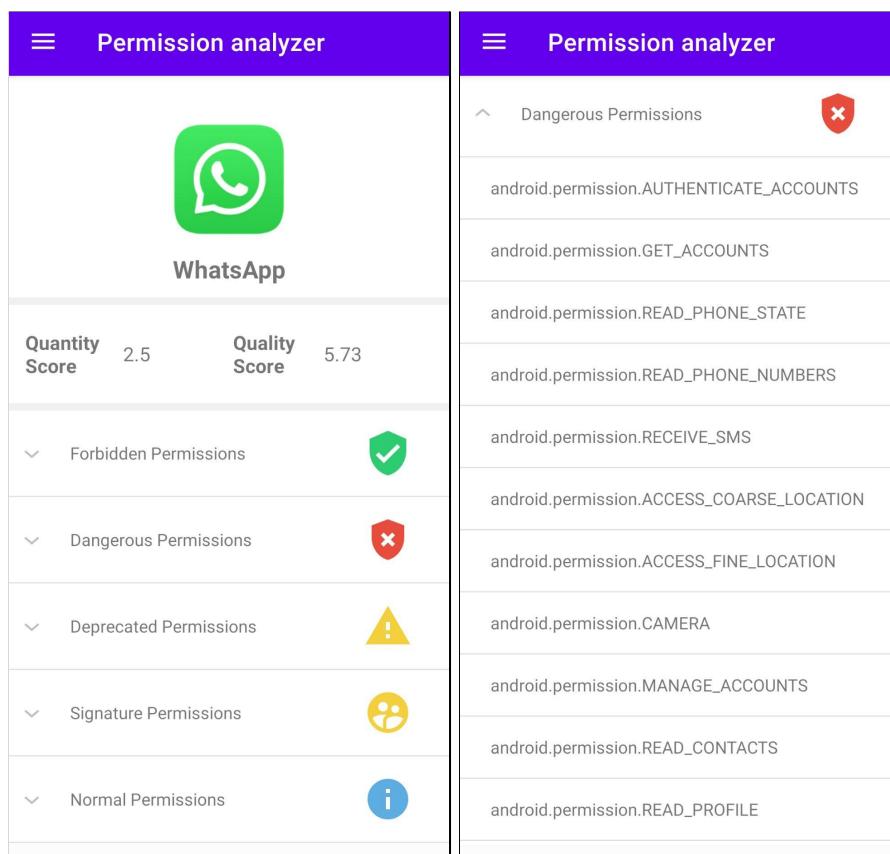


Figure 116: Permission Analyzer - WhatsApp

## - TikTok

- *Figure 117* shows the overall score obtained by analyzing the app with the permission analyzer. As seen, it scored a 2.25 out of 10, which is a very low score. This is due, as we will later see, to its huge number of permissions and the high ratio of unknown and dangerous permissions.

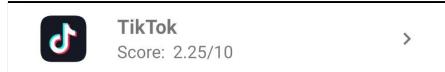


Figure 117: Overall Score - TikTok

- Accessing the result of the analysis, shown in *Figure 118*, it can be seen that the application requests several types of permissions. Specifically, this app requests 9 dangerous permissions, 2 deprecated, 12 normal and 39 unknown, which gives it a rating of 5.08 out of 10 in quality. In terms of quantity, it requests 62 permissions in total, hence it has a 2.25. Of all of them, the most notable permissions are those related to the Camera, Audio, Contacts, Microphone, Phone, Storage and Billing, which make sense within all the functionalities that the app offers. What is quite rare is the great amount of unknown permissions that apparently request to read/write the system settings. Apart from that, it can be appreciated that this application requests a great number of permissions, which could be dangerous in case the application were malicious. In this case, this application is the legitimate one, and those permissions are needed to perform several of the functionalities of the app.

Apps information		Permission analyzer	Permission analyzer
	TikTok		
Quantity Score	2.25	Quality Score	5.08
Forbidden Permissions		Dangerous Permissions	
Dangerous Permissions		android.permission.READ_EXTERNAL_STORAGE	
Deprecated Permissions		android.permission.WRITE_EXTERNAL_STORAGE	
Signature Permissions		android.permission.CAMERA	
Normal Permissions		android.permission.RECORD_AUDIO	
Unknown Permissions		android.permission.READ_CONTACTS	
		com.android.launcher.permission.READ_SETTINGS	
		android.permission.READ_CALENDAR	
		android.permission.WRITE_CALENDAR	
		com.google.android.c2dm.permission.RECEIVE	
		com.huawei.launcher3.permission.READ_SETTINGS	
		com.huawei.launcher3.permission.WRITE_SETTINGS	
		com.huawei.launcher2.permission.READ_SETTINGS	
		com.huawei.launcher2.permission.WRITE_SETTINGS	
		com.ebproductions.android.launcher.permission.READ_SETTINGS	
		com.ebproductions.android.launcher.permission.WRITE_SETTINGS	

Figure 118: Permission Analyzer - TikTok

- **Diccionario de la Lengua Española (DLE)**
  - *Figure 119* shows the overall score obtained by analyzing the app with the permission analyzer. As seen, it scored a 9.6 out of 10, which is a very high

score. This is due, as we will later see, to its impeccable quality and quantity of permissions.

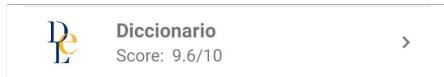


Figure 119: Overall Score - DLE

- Accessing the result of the analysis, shown in *Figure 120*, it can be seen that the application only requests normal permissions, which gives it a rating of 10 out of 10 in quality. In terms of quantity, it only requests two permissions, hence it has a 9.6. Within the functionalities of the application, it makes sense that it uses the Internet and Foreground Service permissions. All in all, it can be appreciated that this application is highly reliable, as it is far from what could be considered a dangerous or suspicious application.

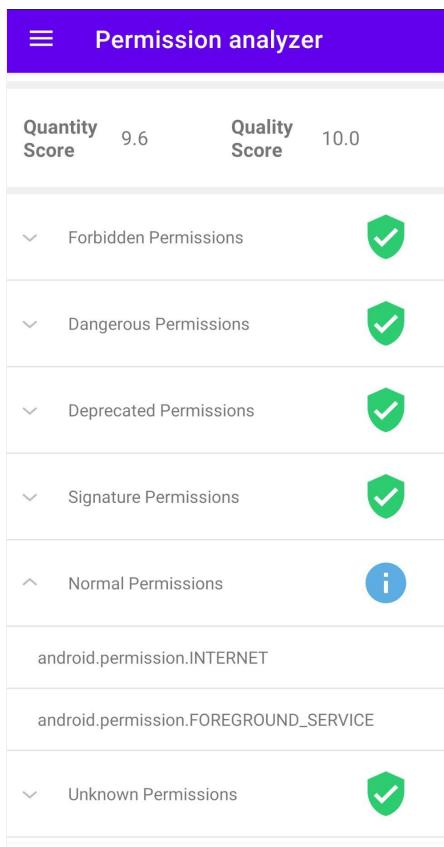


Figure 120: Permission Analyzer - DLE

#### - Nasip Kismet degilimis

- *Figure 121* shows the overall score obtained by analyzing the app with the permission analyzer. As seen, it scored a 8.8 out of 10, which is a very high score. This is due, as we will later see, to the good quality and quantity of permissions.

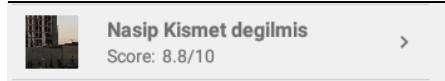


Figure 121: Overall Score - Nasip Kismet degilmis

- Accessing the result of the analysis, shown in *Figure 122*, it can be seen that the application only requests normal and signature permissions, which gives it a rating of 10 out of 10 in quality. In terms of quantity, it only requests six permissions, hence it has a 8.8. Within the functionalities of the application, it should request different permissions, which is suspicious. All in all, it can be appreciated that this application does not request dangerous permissions, but it is suspicious that it does not request the permissions it should need to keep up with its functionality.

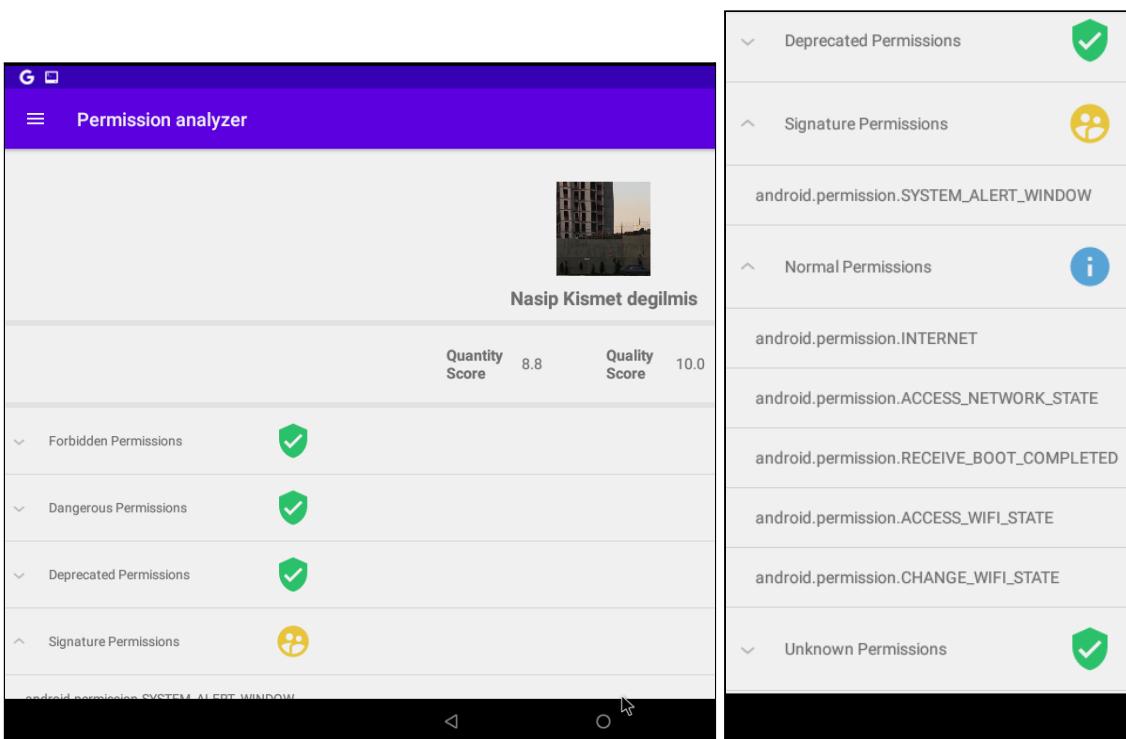


Figure 122: Permission Analyzer - DLE

#### - SmartcardService

- *Figure 123* shows the overall score obtained by analyzing the app with the permission analyzer. As seen, it scored a 4.13 out of 10, which is a very low score. This is due, as we will later see, to the huge number of dangerous permissions it requests.

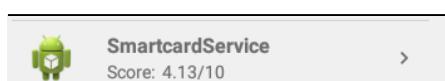
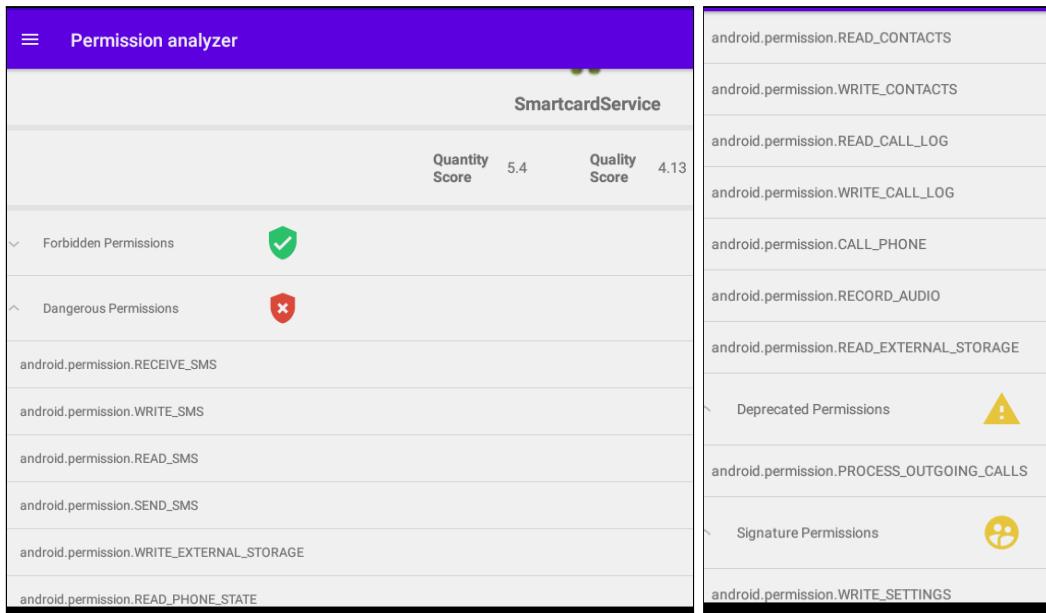


Figure 123: Overall Score - SmartcardService

- Accessing the result of the analysis, shown in *Figure 124*, it can be seen that the application requests several types of permissions. Specifically, this app requests 13 dangerous permissions, 1 deprecated, 1 signature and 8 normal, which gives it a rating of 4.13 out of 10 in quality. In terms of quantity, it requests 23 permissions in total, hence it has a 5.4. Of all of them, the most notable permissions are those related to the Phone, SMS, Call Log, Record Audio and Kill Background Process, which do not make sense within the functionalities that the app allegedly offers. All in all, it can be appreciated that this application requests a great number of permissions, which in this case are extremely dangerous.



*Figure 124: Permission Analyzer - SmartcardService*

## 7.3 Cryptographic Signature Analysis

In this section, the signature analyzer will check the apps mentioned in section 7.1. The result will then be discussed.

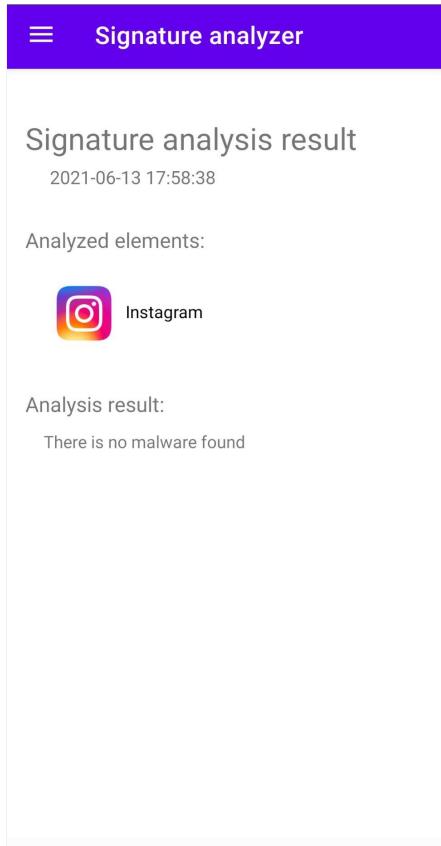
- **Instagram**

- Accessing the Apps Information Fragment, we can see Instagram's signatures (*Figure 125*) in the form of MD5 and SHA-256 hashes.

<b>Sha256</b>
7d08170974591ae8e3a31958682bf2ed6cf49b b69a361b20940a970edb8cc4d7
<b>MD5</b>
a70680dbdf1a2a6e98bc79f3a2626b47

*Figure 125: Hashes - Instagram*

- After performing the signature analysis, the result (*Figure 126*) states that the app is not malicious, as its signature did not match any of the records in the Malware List Table.



*Figure 126: Signature Analyzer - Instagram*

#### - WhatsApp

- Accessing the Apps Information Fragment, we can see WhatsApp's signatures (*Figure 127*) in the form of MD5 and SHA-256 hashes.

Sha256
ae114f30ca041ffff517917f93d40c8d5737aa1 cf5e489bd243d0bae8c19c8e2
MD5
7312a871f65de8dee07cf250501c4e01

*Figure 127: Hashes - WhatsApp*

- After performing the signature analysis, the result (*Figure 128*) states that the app is not malicious, as its signature did not match any of the records in the Malware List Table.

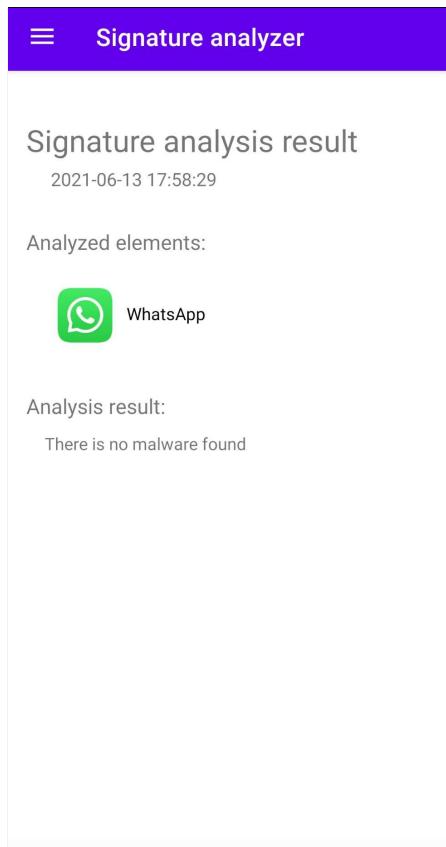


Figure 128: Signature Analyzer - WhatsApp

- **TikTok**

- Accessing the Apps Information Fragment, we can see TikTok's signatures (*Figure 129*) in the form of MD5 and SHA-256 hashes.

Sha256
a328203b1213bd8d5022fd65a3ee11e3f05585
5ff177691298e2cc0097bf57b6
MD5
fa62de7d88f09d3471447ac2b3ad0542

Figure 129: Hashes - TikTok

- After performing the signature analysis, the result (*Figure 130*) states that the app is not malicious, as its signature did not match any of the records in the Malware List Table.

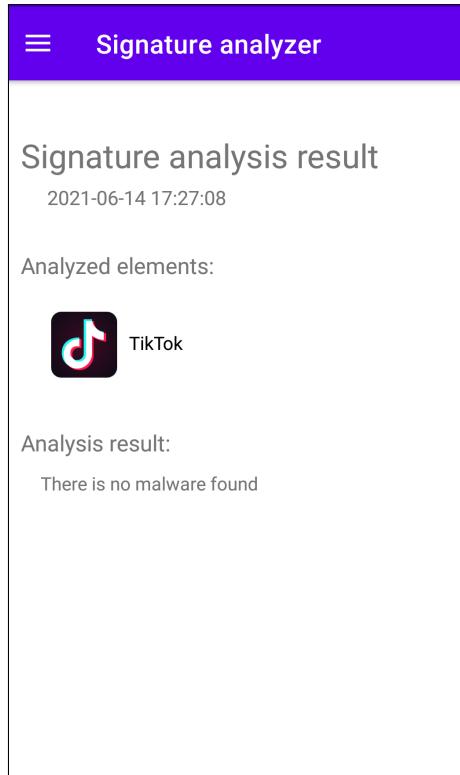


Figure 130: Signature Analyzer - TikTok

- **Diccionario de la Lengua Española (DLE)**

- Accessing the Apps Information Fragment, we can see DLE's signatures (*Figure 131*) in the form of MD5 and SHA-256 hashes.

Sha256	a8d579bfeea87f805170de3d2c16b2425a9b93 8d1dca4e144ee747ccdbb44f94
MD5	4547badc463c565d6ddd003ca088f2e0

Figure 131: Hashes - DLE

- After performing the signature analysis, the result (*Figure 132*) states that the app is not malicious, as its signature did not match any of the records in the Malware List Table.

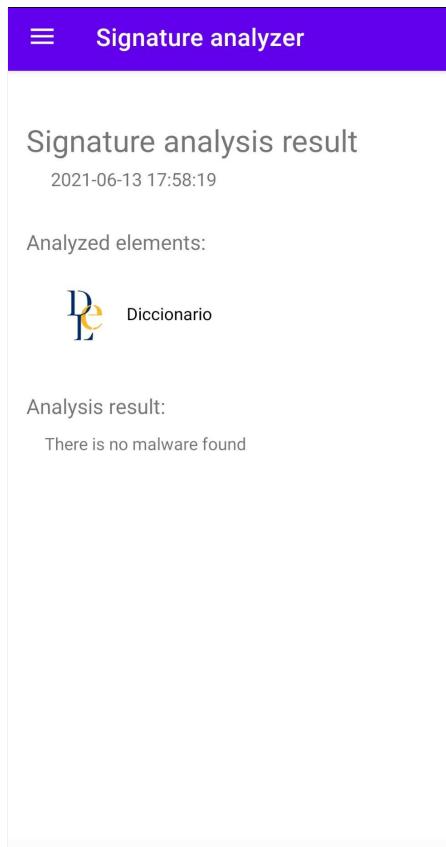


Figure 132: Signature Analyzer - DLE

#### - Nasip Kismet degilmis

- Accessing the Apps Information Fragment, we can see WhatsApp's signatures (*Figure 133*) in the form of MD5 and SHA-256 hashes.

```
Sha256  
fc4f2c2f88e4366fc19d1350cc54f3e5c0f083e3  
3238f784341b6ff5dcee3ce1  
  
MD5  
f2efdb765feaccd7c1697742d7542803
```

Figure 133: Hashes - Nasip Kismet degilmis

- After performing the signature analysis, the result (*Figure 134*) states that the app is not malicious, as its signature did not match any of the records in the Malware List Table. This is due to the fact that our Malware List Table has a limited number of signatures and, in this case, the signature of this malware is not in any of its records.

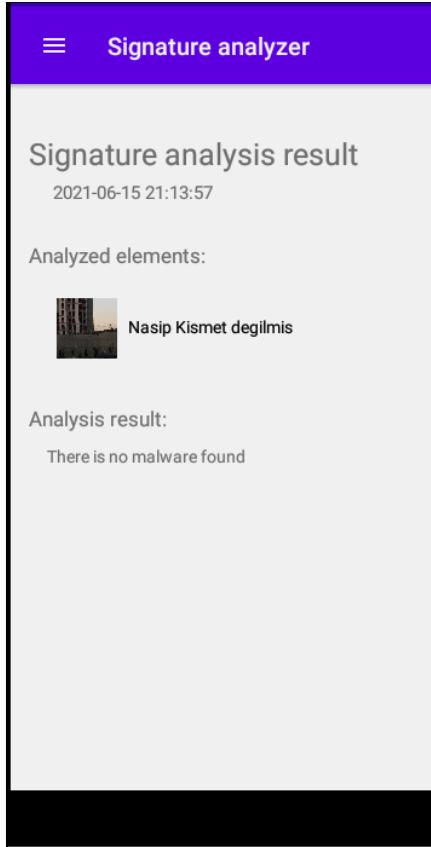


Figure 134: Signature Analyzer - Nasip Kismet degilmis

- **SmartcardService**

- Accessing the Apps Information Fragment, we can see WhatsApp's signatures (*Figure 135*) in the form of MD5 and SHA-256 hashes.

Sha256
a328203b1213bd8d5022fd65a3ee11e3f05585
5ff177691298e2cc0097bf57b6
MD5
fa62de7d88f09d3471447ac2b3ad0542

Figure 135: Hashes - SmartcardService

- After performing the signature analysis, the result (*Figure 136*) states that the app is not malicious, as its signature did not match any of the records in the Malware List Table. This is due to the fact that our Malware List Table has a limited number of signatures and, in this case, the signature of this malware is not in any of its records.

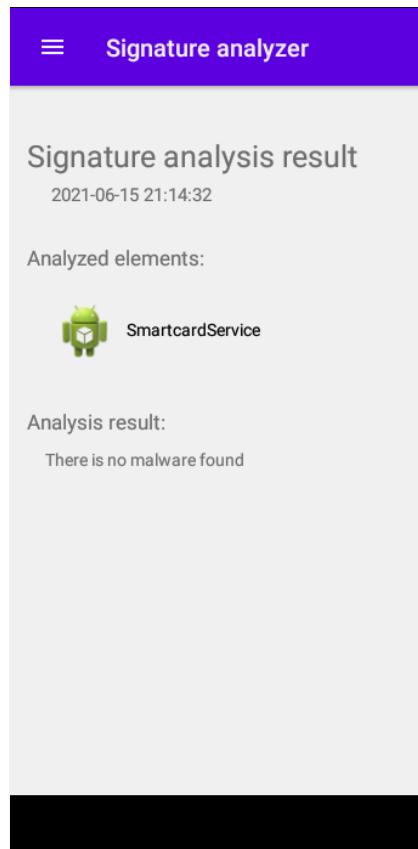


Figure 136: Signature Analyzer - SmartcardService

## 7.4 Heuristic Log Analysis

In this section, the log analyzer will analyze the apps mentioned in section 7.1. The result will then be discussed.

### - Instagram

- While the log analyzer was collecting the logs, we accessed Instagram and interacted a bit with the main page, looking at recent posts and some stories. We then tried to take a photo, for which it asked for permission to access the camera which we decided not to grant. After that, we accessed Directs and tried to send an audio to a contact, for which it asked for permission to access the microphone which we decided not to grant. Finally, we saw some of the posts and Reels that appeared in the Explore tab and accessed a specific account after searching for it in the search bar.
- After we were done, we stopped the analysis and the results were obtained (*Figure 137*). It can be seen that during the execution of the analysis, four applications were running: Instagram, AndroidMalwareAnalyzer, WhatsApp and Stickers. WhatsApp probably had some service running in the background, so it is not surprising to see that it generated some logs. As for the Stickers app, perhaps it would be interesting for the user to analyze it separately since it should not be running, but that is not the scope of this

analysis. Four target elements have been detected during the execution of Instagram: Camera, Messaging, Microphone and Internet. Since there are a low number of logs related to each target element, the user should not worry, as they are probably false positives or, having been asked to grant permissions to the camera and microphone, it may have generated some logs related to them. For this reason, finding that the app has no access to the camera and microphone should not be alarming.

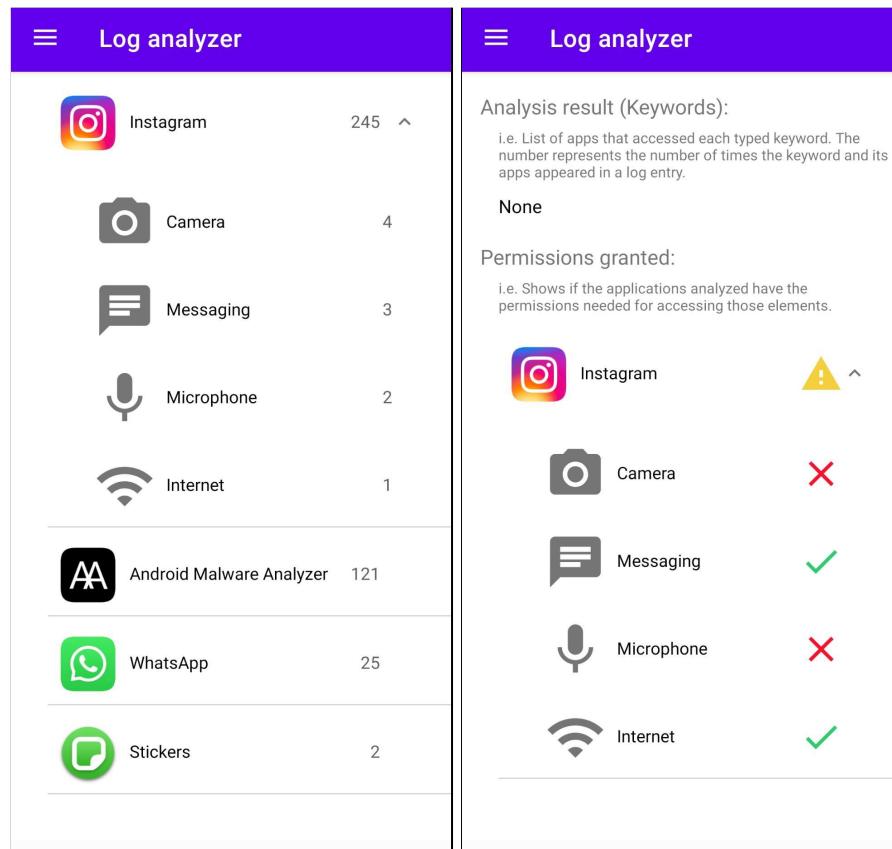


Figure 137: Log Analyzer - DLE

- Reading the logs that were stored on the server, we saw that Camera was triggered when the camera activity was loaded: “06-15 17:07:55.469 1375 8410 v windowmanager: adding window{a67672 u0 khcd.4zp.reel\_composer\_camera} to window{620175e u0 com.instagram.android/com.instagram.mainactivity.mainactivity}”. Regarding the Microphone, the logs were generated when we were prompted to grant the permission: “06-15 17:08:03.273 31338 31338 v grantpermissionsactivity: logged buttons presented and clicked permissiongroupname=android.permission-group.microphone uid=10278 package=com.instagram.android presentedbuttons=25 clickedbutton=8”

- From this analysis we can conclude that this application performs the promised functionalities and does not attempt to do anything outside of its supposed behaviour.
- **WhatsApp**
  - While the log analyzer was collecting the logs, we accessed a WhatsApp conversation to take a photo and send it. We also activated the microphone and sent a document stored in the device. Finally, we made a video call with another contact.
  - After we were done, we stopped the analysis and the results were obtained (*Figure 138*). It can be seen that during the execution of the analysis, three applications were running: WhatsApp, AndroidMalwareAnalyzer and Instagram. Instagram probably had some service running in the background, so it is not surprising to see that it generated some logs. Five target elements have been detected during the execution of Instagram: Camera, Microphone, Messaging, SDcard and Internet. As the number of logs regarding the microphone and camera, their numbers are high enough to be sure that those have been accessed.

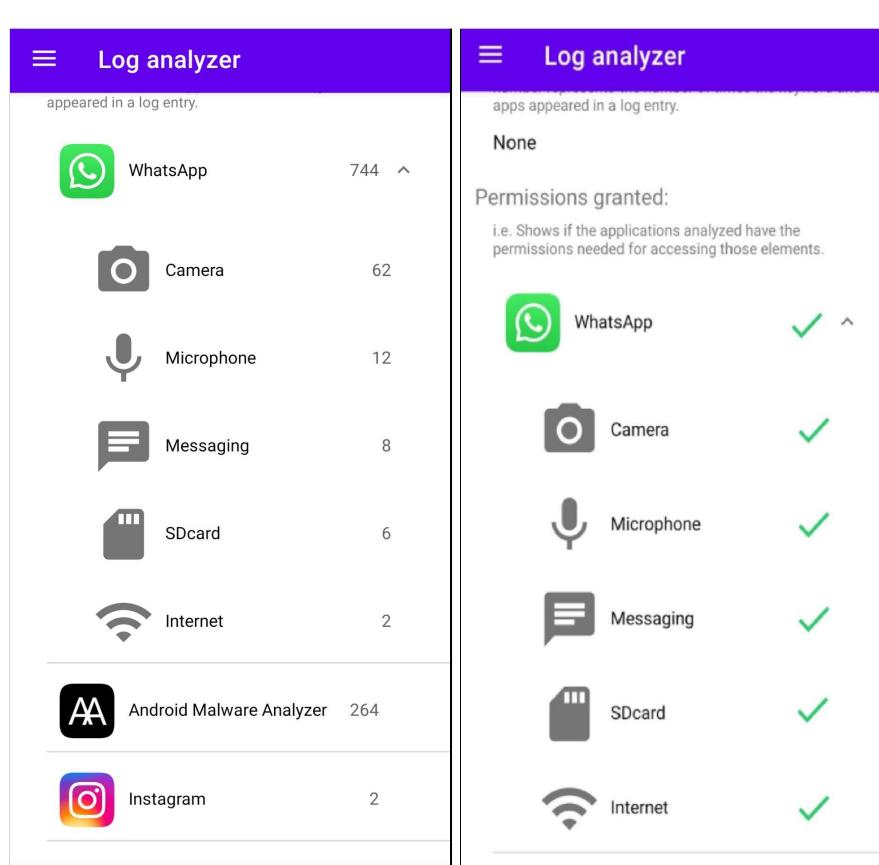


Figure 138: Log Analyzer - WhatsApp

- Reading the logs that were stored on the server, we saw that the Camera logs were generated when the camera was opened and when the camera activity

was called. As an example, we show this log: "06-15 19:33:12.549 14324 15103 i camera : open camera: 1, package name: com.whatsapp". Regarding the SDcard, the logs were generated when we searched the locally stored documents to send one of them: "06-15 19:33:12.444 14324 14324 w com.whatsapp: type=1400 audit(0.0:10590): avc: granted { setattr } for pid=14324 name="/" dev="sdcardfs" ino=11048 scontext=u:r:untrusted\_app:s0:c147,c256,c512,c768 tcontext=u:object\_r:sdcardfs:s0 tclass=filesystem".

- From this analysis we can conclude that this application performs the promised functionalities and does not attempt to do anything outside of its supposed behaviour.
- **TikTok**
  - While the log analyzer was collecting the logs, we accessed TikTok and interacted a bit with the main page, looking at recent posts. We then recorded a video, for which it asked for permission to access the camera which we decided to grant.
  - After we were done, we stopped the analysis and the results were obtained (*Figure 139*). It can be seen that during the execution of the analysis, just five applications were running: TikTok, AndroidMalwareAnalyzer, Zity, Wible and Instagram. Regarding Instagram, it probably had some service running in the background, so it is not surprising to see that it generated some logs. As for Zity and Wible apps, perhaps it would be interesting for the user to analyze it separately since it should not be running, but that is not the scope of this analysis. Four target elements have been detected during the execution of TikTok: Camera, Messaging, Location and SDCard. As it concerns the Camera, it makes sense to find logs because during the analysis it has been used. Regarding the rest of the logs, since there are a low number of logs related to each target element, the user should not worry as they are probably false positives. For this reason, finding that the app has no access to the location should not be alarming.

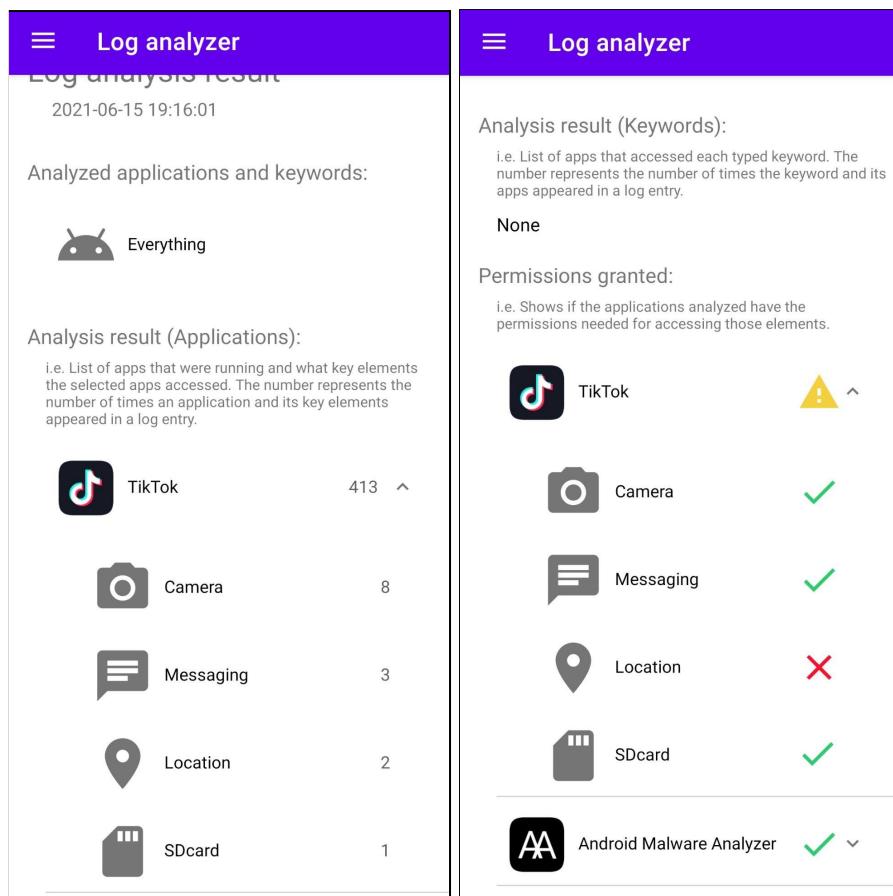


Figure 139: Log Analyzer - TikTok

- Reading the logs that were stored on the server, we saw that Camera was triggered when the camera was opened: "06-15 19:14:15.738 31375 32062 i cameramanager: open camera: 1, package name: com.zhiliaoapp.musically".
- As regards the SDcard, the logs were: "06-15 19:14:57.411 31375 31375 w frescoioboundex: type=1400 audit(0.0:7330975): avc: granted { read open } for pid=31375 path="/storage/emulated/0/android/data/com.zhiliaoapp.musically/cache/picture/fresco\_cache/v2.ols100.1/35/eq\_mss\_q\_qyzoxfxqyavkx5b5wjw.cnt" dev="sdcarddfs" ino=254257 scontext=u:r:untrusted\_app:s0:c99,c257,c512,c768 tcontext=u:object\_r:sdcarddfs:s0 tcclass=file"
- As to the Location, the logs found were "06-15 19:14:12.993 2089 2467 i pg\_ash : unp\_gps:com.zhiliaoapp.musically uid:10355 result:true".
- Finally, regarding to the SDCard, the logs found were "06-15 19:14:12.991 2009 5099 d assistantservice-1030200: handlemessage app switch"

```
frompackage:com.huawei.android.launcher,  
topackage:com.zhiliaooapp.musically".
```

- From this analysis we can conclude that this application performs the promised functionalities and does not attempt to do anything outside of its supposed behaviour.
  
- **Diccionario de la Lengua Española (DLE)**
  - While the log analyzer was collecting the logs, we accessed the DLE application and did a couple of searches for the meanings of different words. We also accessed two of the links in the application, which redirect to the default browser to display information from the RAE and the app.
  - After we were done, we stopped the analysis and the results were obtained (*Figure 140*). It can be seen that during the execution of the analysis, three applications were running: DLE, AndroidMalwareAnalyzer and Firefox Focus. Firefox Focus appears because the DLE application links redirected the user to the default browser, which in the case of the device on which we tested the analysis, was Firefox Focus. Two target elements have been detected during the execution of the DLE application: Messaging and Storage. Since there are a low number of logs related to each target element, the user should not worry, as they are probably false positives. For this reason, finding that the app has no access to the storage should not be alarming.

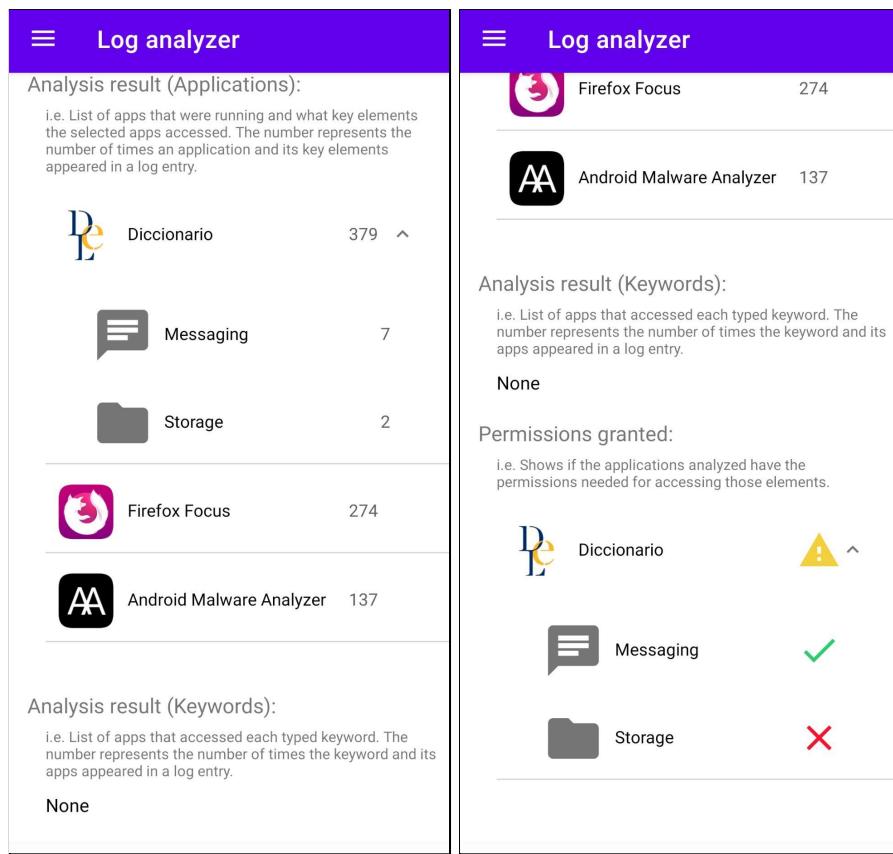


Figure 140: Log Analyzer - DLE

- Reading the logs that were stored on the server, we realized that Messaging was triggered because when switching from the DLE application to Firefox Focus and back, several logs like "06-13 17:40:41.598 1975 4130 d assistantservice-1030200: handlemessage app switch frompackage:com.huawei.android.launcher, topackage:es.rae.dle" were generated, which is detected as Messaging by a false positive. Storage has been detected because two logs were created in this form "06-13 17:40:31.424 13742 13760 i hwapicachemangerex: apicache path=/storage/emulated/0 state=mounted key=es.rae.dle#10122#".
- From this analysis we can conclude that this application performs the promised functionalities and does not attempt to do anything outside of its supposed behaviour.

#### - Nasip Kismet degilmis and SmartcardService

- Due to their malicious nature, it is necessary to perform an analysis of these two applications in a secure environment such as a virtual machine. When trying to perform the analysis, we realized that the applications crashed each time they were opened, probably because they were programmed to work in

an older Android version. For this reason, we were unable to perform the log analysis for these applications.

## 8. Individual Work

This chapter summarises the contributions of each participant in this project, listing everything learned in the process of carrying it out. The work has been carried out jointly and fairly, dividing the work into each of the points that make it up. These are the cryptographic signatures analysis, the analysis of logs and the analysis of permissions.

### 8.1 Daniel Puente Arribas

My work has had more weight in the analysis of permissions.

The beginning of my work consisted of a global study of the metadata contained within the `PackageInfo` object. In this way, I learned about its uses and characteristics to understand how to obtain the data and say which fields were the most useful and relevant to develop our analysis.

Next, I began the development of the `Apps Information` fragment and my first approach to the logical and visual development of the application consisted in showing the list of all the installed applications, as well as the relevant information of each one.

Once this was done, I continued with the development of the fragment that shows the details and information for each application. During this stage, I started to work with the requested permissions and I saw the need to create a view dedicated to the permissions in order to classify them according to the scope of restricted data that the apps can access, and the scope of restricted actions that apps can perform when the system grants them permission.

This way, before getting into the permissions analysis, I did a deep research about Android permissions and learned how to declare permissions, where to declare them, how to access them and how to classify them according to the domain which they belong to and according to the level of access they request to the device.

Subsequently, I built a dataset composed of permissions and permission groups out of various sources with the objective of using it later in permissions analysis. The interesting thing about this dataset is that it turned out to be very complete and very useful when it comes to correctly classifying the permissions. After all the information had been collected, the dataset information was added as tables to the application's SQLite database.

Once I obtained the necessary knowledge about Android permissions and having created a dataset that stores them, I began to develop the `Permission Analyzer` fragment. To do this, I came up with two algorithms that assign two different ratings in order to evaluate applications based on the permissions they request. Actually, these two ratings provide two very interesting views on applications; a quantitative view and another deeper and qualitative view.

After I have finished designing the logical and functional part of the permissions analyzer, I implemented it in the application and I continued developing the GUI of the analyzer.

Regarding this document, I wrote the permission analyzer sections of chapters 4, 5, 6 and 9. We also divided chapter 3, section 1.4 and the abstract equally. I also wrote my corresponding section of chapter 8 and sections 1.1 and 1.3. Lastly, I was in charge of correctly citing the bibliography.

To conclude, my main contributions in this project focus on the analysis of applications metadata, the research of Android permissions and in relation to the extraction of information from the apps and the analysis of permissions, elaborating the functional and logic design as well as their user interface design and its corresponding implementation.

## 8.2 José Ignacio Daguerre Garrido

I created the local database with its two tables of malware signatures and existing Android permissions, which is essential to perform the static analysis. I was in charge of creating an EC2 instance of AWS to be able to monitor the logs sent from an Android device in order to eliminate the cost of doing it locally. I was also in charge of establishing a connection between this instance and a TCP server, which connected remotely with the application developed in Android Studio.

On the other hand, I have participated in the implementation of the Cryptographic Signature Analysis fragments, being one of the three fundamental pillars that make up the whole application.

Regarding this document, I wrote the signature analyzer sections of chapters 4, 5, 6 and 9. We also divided chapter 3, section 1.4 and the abstract equally. I also wrote my corresponding section of chapter 8 and section 1.2. Alongside Ramón, we wrote chapter 2.

I have been able to understand not only the internal structure of Android applications but also how Linux is composed and the security services and structure that Android provides.

Despite the lack of knowledge of the Android Studio tool due to its complex structure and use, thanks to this project I have been able to deal with this tool and clearly understand its usability, as well as discovering the benefits and implementations that can be applied to the development of Android applications. As Android Studio is programmed in Java, I have been able to strengthen my Java programming skills and become more fluent.

I have also learned how to make connections between AWS instances and mobile devices through a TCP server using sockets. I have found Spark Streaming really exciting, which is responsible for processing data in real time, in our case the logs of an Android device. It is going to be helpful to implement applications that offer greater efficiency and performance.

It is undoubtedly the case that we live in an age when people are unaware of the security and structure of their mobile device and the risks involved. What I have learnt is that being knowledgeable about how application permissions work gives you a certain basic understanding of how to deal with malware.

## 8.3 Ramón Costales de Ledesma

First of all, I researched some papers and documents regarding the topic of malware detection in Android. This allowed me to plan what we wanted to achieve with this project. From this research, we decided that we wanted to develop an app that would perform the three types of analyses we have been able to achieve.

I had some previous understanding about Android internals, but when it came to Android programming, the only previous knowledge I had was how to program in Java. For this reason, I had to study how Android applications were developed and how to use Android Studio.

Once we all were familiar with Android programming, we decided our next job was to jointly develop an application that would pull the logs from an Android device and send them to an instance of AWS EC2 that we used as the server. After we managed to connect them, we developed a Spark Streaming script that processed those logs. Finally, I programmed a bash script that retrieved the results and sent them to the application.

Once we finished that, we decided to divide the three methods of analysis between the three components of the team. I decided to create a template of the application using fragments so that programming the app in parallel would be straightforward.

The analysis method that I had to develop was the Log Analysis. Starting from what we already had, I expanded the functionality until I reached the Log Analyzer Fragment that we now have. I created the Server Settings Fragment to store the addresses and control which address to use to connect to the server. I developed the Previous Result Fragment to store the results of both log and signature analysis. I developed the Home Fragment and About Us Fragment.

Finally, I also had to develop the server. For test purposes, I created a new EC2 instance. After the configuration, I created a C script that allowed multi connection to the server. In the end, the C script now manages connections, processes the logs, sends them to the Spark Streaming script and then returns the results back to the application. I also further developed the Spark Streaming script to meet our new requirements.

Regarding this document, I wrote the log analyzer sections of chapters 4, 5, 6 and 9 (I also did the chapter 6 sections that I developed, mainly Home Fragment, Server Settings and About Us). We also divided chapter 3, section 1.4 and the abstract equally. I also wrote my corresponding section of chapter 8 and section 1.5. I also gave an introduction to most of the chapters. Alongside José, we wrote chapter 2. Lastly, I performed the analysis shown in chapter 7 and wrote its results.

# 9. Conclusions & Future Work

This chapter reviews the work carried out by the team and the result of the project. It also discusses some improvements the team didn't have the time to carry out for the three types of analysis and brainstorms future work that could be done to give a suitable closure to the application.

## 9.1 Overview

At the end of the project and after having exposed the necessity of an anti-malware software installed on the device of every user, we believe that our application can fill some of the crucial needs that every user of a mobile phone requires.

We believe that the objectives set at the beginning of the project have been met. We have developed an application that gives the user plenty of information that can hugely help determine if an installed app is malware or at the very least if it has suspicious behaviour.

Not only have we managed to develop three different types of analysis that have proven to be extremely useful, but we also have programmed a way to display general information on the installed apps. In addition, thanks to saving the previous results, the user can track the applications behaviour in different periods of time, allowing the user to get an overall image of the installed apps behaviour.

We also think that the application will encourage users to actively analyze their applications and that they will learn the importance of ensuring the veracity of the applications being installed based on their permissions and their behaviour.

Summarizing our perspective of the project, we believe that the work we have carried out throughout this year has been worthwhile and that the application has turned out to be a success, even if there is still plenty of work to do and some things to polish. In fact, we plan to keep using it in our daily lives from now on. We also appreciate all the new knowledge that we have grasped thanks to the research we have done throughout this year.

## 9.2 Applied Knowledge

We have to acknowledge certain subjects taken during the *Ingeniería Informática* Degree that have facilitated the development of our project:

- Thanks to the knowledge acquired in the subjects **Bases de Datos** (BD) and **Ampliación de Bases de Datos** (ABD), we have been able to build the *MalwareDB* and *PrevResultsDB* databases properly and to structure them efficiently.
- The **Cloud y Big Data** (CLO) course was a great help for the implementation of the AWS EC2 instance, since we obtained the necessary knowledge to be able to handle the instance and its respective services. It also helped us to develop the Spark Streaming script, which is responsible for processing the logs sent by Android devices in real time.

- As for the programming language of the application we developed, the subject **Tecnologías de la Programación** (TP) had already given us the proficiency necessary to program in Java without having to learn the programming language from scratch.
- Thanks to the **Redes** (RED) subject, we knew the basics of TCP connections, which helped us perform the connection between the app and the server.
- When it came to programming the TCP server script used to connect the app with the AWS EC2 instance, the **Ampliación de Sistemas Operativos y Redes** (ASOR) subject provided us with the sufficient and necessary knowledge for such an implementation.
- The C script used in the server was also developed with the C programming skills obtained from the subjects **Ampliación de Sistemas Operativos y Redes** (ASOR), **Sistemas Operativos** (SO), **Arquitectura de Linux y Android** (LIN) and **Fundamentos de la Programación** (FP).
- The benefit of having taken the **Sistemas Operativos** (SO) and **Arquitectura de Linux y Android** (LIN) courses was the understanding of the internal aspects of the Android operating system.
- A subject that has been vital in the development of the Cryptographic Signature Analysis section is **Redes y Seguridad I & II** (RS1 and RS2), which not only provided us with the knowledge and applications of hash algorithms, but we also learned about malware, which is a fundamental part of this project, and taught us some key concepts that have helped us tremendously while making this project.
- It is also important to mention that every subject that taught us how to use Linux (**Sistemas Operativos** (SO), **Arquitectura de Linux y Android** (LIN), **Redes** (RED) and **Redes y Seguridad II** (RS2)) helped configure the cloud instance.
- Thanks to all the team projects done throughout the career, we three already knew how to use GitHub Desktop and were very efficient when working in parallel.

## 9.3 Improvements

This section reflects various improvements of the different analyzers that we have not had enough time to carry out despite having the knowledge to be able to implement them.

### 9.3.1 Signature analyzer

- Expand the malware dataset of digital signatures (hashes), as well as being able to store for each database row different algorithms of hashing such as SHA512, SHA256, etc applied on a single malware.
- Migrate the database to Cloud so that the devices that have installed our application do not need to have it locally, thereby taking up less space and having higher performance.

- Create another database or modify and extend the existing one to contain the hashes of the top 100 (or more) most used/installed apps in the Google Play Store in order to compare the hashes of the apps installed on your mobile with those in the database to detect malware on your Android device.

### 9.3.2 Log analyzer

- Configure the server so that the scripts can be running all the time. Right now, the only way to achieve this is by having the scripts running in a shell that is connected to the server through ssh, which is not ideal.
- In the results, show the time when each target element was accessed.
- Perform the connection to the server through SSL.
- Change the *AsyncTask* for an *Executor* or a *Service*, since the *AsyncTask* is deprecated and can be unreliable sometimes.
- Ensure that the connection is performed from an *AndroidMalwareAnalyzer* application to prevent malicious connections.
- Use the log history we have created on the server to allow users to query their logs and search by date or application.

### 9.3.3 Permission analyzer

- Store in the database all the application permissions and, in each analysis, look for any changes in the permissions in order to warn the user. Finally, keep updated the database with these changes.
- Reduce the analysis time by creating new threads.
- Get all the application categories acknowledged by the Google Play Store in order to be more precise when trying to predict the volume of permissions. This can be done, for example, implementing *android-market-api* [96].
- Improve the navigability and the GUI.

## 9.4 Future Work

This work has been carried out over approximately six months, which is the expected time for the development of this project. However, the work done here can be improved by extending its utilities as indicated below.

### 9.4.1 Signature analyzer

- Create a reporting option for the user in case she/he detects a new malicious application by using other analysis methods (Log analyzer or Permission analyzer) or from another source, in order to generate its hash and include it in our database.

### 9.4.2 Log analyzer

- Implement machine learning for contrasting logs of specific applications between different users, which would allow the application to detect deviant behaviour.

- Improve the analysis to avoid false positives when detecting target elements.
- Develop an automatic service that launches the log analysis every certain period of time (for example, every hour for 5 minutes) and briefs the user of the results obtained from the last time the application was accessed.

#### 9.4.3 Permission analyzer

- Keep updated and complete the dataset of permissions and domains with those still unknown by the application.

# Bibliography

- [1] *Android*. Retrieved from  
<https://www.android.com/>
- [2] CCN-CERT IA-04/19. (2019, January). *Informe Anual 2018 Dispositivos y comunicaciones móviles*. Retrieved from  
<https://www.ccn-cert.cni.es/informes/informes-ccn-cert-publicos/3464-ccn-cert-ia-04-19-informe-anual-2018-dispositivos-moviles/file.html>
- [3] Yarow, Jay. (2014, March 28). Insider. *Android's share of the computing market*. Retrieved from  
<https://www.businessinsider.com/androids-share-of-the-computing-market-2014-3>
- [4] Wikipedia. *Comparison of antivirus software*. Retrieved from  
[https://en.wikipedia.org/wiki/Comparison\\_of\\_antivirus\\_software](https://en.wikipedia.org/wiki/Comparison_of_antivirus_software)
- [5] Kaspersky. (2021, March 1). Virología móvil 2020. Retrieved from  
<https://securelist.lat/mobile-malware-evolution-2020/93058/>
- [6] D. Wu, C. Mao, T. Wei, H. Lee and K. Wu, "DroidMat: Android Malware Detection through Manifest and API Calls Tracing," 2012 Seventh Asia Joint Conference on Information Security, 2012, pp. 62-69. Retrieved from <https://ieeexplore.ieee.org/document/6298136>
- [7] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon & Konrad Rieck. (2014). *DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket*. University of Gottingen, Germany & Siemens CERT, Munich, Germany. Retrieved from  
<https://www.sec.cs.tu-bs.de/pubs/2014-ndss.pdf>
- [8] Z. Zhao and F. C. Colon Osono, "'TrustDroid™': Preventing the use of SmartPhones for information leaking in corporate networks through the used of static analysis taint tracking," 2012 7th International Conference on Malicious and Unwanted Software, 2012, pp. 135-143, doi: 10.1109/MALWARE.2012.6461017. Retrieved from  
<https://ieeexplore.ieee.org/document/6461017>
- [9] Desnos, Anthony. *Androguard: Reverse engineering, Malware and goodware analysis of Android applications*. Retrieved from <https://github.com/androguard/androguard>
- [10] Rosenzwig, Eran & Asis, Gili. (2008, December 31). *ANDROMALY – ANOMALY DETECTION IN ANDROID PLATFORM*. Retrieved from <https://andromaly.wordpress.com/>
- [11] A. Saracino, D. Sgandurra, G. Dini and F. Martinelli, "MADAM: Effective and Efficient Behavior-based Android Malware Detection and Prevention," in IEEE Transactions on Dependable and Secure Computing, vol. 15, no. 1, pp. 83-97, 1 Jan.-Feb. 2018, doi: 10.1109/TDSC.2016.2536605. Retrieved from <https://ieeexplore.ieee.org/document/7422770>
- [12] Iker Burguera, Urko Zurutuza & Simin Nadjm-Tehrani. (2011, October 17). *Crowdroid: Behavior-Based Malware Detection System for Android*. Electronics and Computing Department,

Mondragon University & Dept. of Computer and Information Science Linköping University.  
Retrieved from <https://www.ida.liu.se/labs/rtslab/publications/2011/spsm11-burguera.pdf>

[13] Liu L., Yan G., Zhang X., Chen S. (2009) VirusMeter: Preventing Your Cellphone from Spies. In: Kirda E., Jha S., Balzarotti D. (eds) Recent Advances in Intrusion Detection. RAID 2009. Lecture Notes in Computer Science, vol 5758. Springer, Berlin, Heidelberg. Retrieved from [https://link.springer.com/chapter/10.1007/978-3-642-04342-0\\_13](https://link.springer.com/chapter/10.1007/978-3-642-04342-0_13)

[14] T. Bläsing, L. Batyuk, A. Schmidt, S. A. Camtepe and S. Albayrak, "An Android Application Sandbox system for suspicious software detection," 2010 5th International Conference on Malicious and Unwanted Software, 2010, pp. 55-62, doi: 10.1109/MALWARE.2010.5665792. Retrieved from <https://ieeexplore.ieee.org/document/5665792>

[15] Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, Michalis Faloutsos. (2012, August). *ProfileDroid: multi-layer profiling of android applications*. Mobicom '12: Proceedings of the 18th annual international conference on Mobile computing and networking, pp. 137-148. Retrieved from <https://dl.acm.org/doi/10.1145/2348543.2348563>

[16] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, Tao Xie. (2013, August). *WHYPER: Towards Automating Risk Assessment of Mobile Applications*. North Carolina State University. Retrieved from [https://www.usenix.org/system/files/conference/usenixsecurity13/sec13-paper\\_pandita.pdf](https://www.usenix.org/system/files/conference/usenixsecurity13/sec13-paper_pandita.pdf)

[17] Oficina de Seguridad del Internauta. *CONAN mobile*. Retrieved from <https://www.osi.es/es/conan-mobile>

[18] Zimperium. *New Advanced Android Malware Posing as “System Update”*. Retrieved from <https://blog.zimperium.com/new-advanced-android-malware-posing-as-system-update/>

[19] Google Security Blog. *PHA Family Highlights: Bread (and Friends)*. Retrieved from <https://security.googleblog.com/2020/01/pha-family-highlights-bread-and-friends.html>

[20] Wikipedia. *SIM Swapping*. Retrieved from [https://es.wikipedia.org/wiki/SIM\\_swapping](https://es.wikipedia.org/wiki/SIM_swapping)

[21] Android Developers. *Android Platform Architecture*. Retrieved from <https://developer.android.com/guide/platform>

[22] Wikipedia. *Android application package*. Retrieved from [https://en.wikipedia.org/wiki/Android\\_application\\_package](https://en.wikipedia.org/wiki/Android_application_package)

[23] Ratazzi, Paul. (2016, December). *Understanding and Improving Security of the Android Operating System*. Retrieved from [https://www.researchgate.net/figure/Figure-APK-file-structure\\_fig2\\_316793316](https://www.researchgate.net/figure/Figure-APK-file-structure_fig2_316793316)

[24] Android Developers. *App Manifest Overview*. Retrieved from <https://developer.android.com/guide/topics/manifest/manifest-intro>

[25] Android Developers. *Permissions on Android*. Retrieved from <https://developer.android.com/guide/topics/permissions/overview>

- [26] Android Developers. *Manifest.permission\_group*. Retrieved from [https://developer.android.com/reference/android/Manifest.permission\\_group](https://developer.android.com/reference/android/Manifest.permission_group)
- [27] Android Developers. *Privacy changes in Android 10*. Retrieved from <https://developer.android.com/about/versions/10/privacy/changes>
- [28] Android Developers. *Permissions updates in Android 11*. Retrieved from <https://developer.android.com/about/versions/11/privacy/permissions>
- [29] Sinhal, Ankit. (2017, April 5). Android Java Point. *Closer Look At Android Runtime: DVM vs ART*. Retrieved from <http://androidjavapoint.blogspot.com/2017/04/closer-look-at-android-runtime-dvm-vs.html>
- [30] Yaghmour, Karim. (2013, March 6). Slideshare. Embedded Android Workshop at Linaro Connect Asia 2013: *Native Android Userspace*. Retrieved from <https://www.slideshare.net/opersys/native-android-userspace-part-of-the-embedded-android-workshop-at-linaro-connect-asia-2013>
- [31] Android Developers. *Application Fundamentals*. Retrieved from <https://developer.android.com/guide/components/fundamentals>
- [32] Android Developers. *Intents and Intent Filters*. Retrieved from <https://developer.android.com/guide/components/intents-filters>
- [33] Trend Micro. (2012, July 20). *A Look at Google Bouncer*. Retrieved from [https://www.trendmicro.com/en\\_us/research/12/g/a-look-at-google-bouncer.html](https://www.trendmicro.com/en_us/research/12/g/a-look-at-google-bouncer.html)
- [34] Google Play Protect. Retrieved from <https://developers.google.com/android/play-protect/>
- [35] Google Play Protect. *On-device protections*. Retrieved from <https://developers.google.com/android/play-protect/client-protections>
- [36] Google Play Protect. *Cloud-based protections*. Retrieved from <https://developers.google.com/android/play-protect/cloud-based-protections>
- [37] AV-TEST. (2018, May). *16 Android Security Apps vs. Google Play Protect in an Endurance Test*. Retrieved from <https://www.av-test.org/en/news/16-android-security-apps-vs-google-play-protect-in-an-endurance-test/>
- [38] AV-TEST. (2021, March). *AV-TEST Product Review Report*. Retrieved from <https://www.av-test.org/en/antivirus/mobile-devices/android/march-2021/google-play-protect-24.3-213208/>
- [39] Android Developers. *Android 6.0 Changes*. Retrieved from <https://developer.android.com/about/versions/marshmallow/android-6.0-changes>
- [40] G DATA. (2018, July 11). *Cyber attacks on Android devices on the rise*. Retrieved from <https://www.gdatasoftware.com/blog/2018/11/31255-cyber-attacks-on-android-devices-on-the-rise>

- [41] Aguilar, Ricardo. (2020, March 9). Xataka Android. *Android encabeza la lista de los sistemas operativos más vulnerables de los últimos 20 años*. Retrieved from <https://www.xatakandroid.com/sistema-operativo/android-encabeza-lista-sistemas-operativos-vulnerables-ultimos-20-anos>
- [42] CVE Details. (2021). *Google Android: CVE security vulnerabilities, versions and detailed reports*. Retrieved from [https://www.cvedetails.com/product/19997/Google-Android.html?vendor\\_id=1224](https://www.cvedetails.com/product/19997/Google-Android.html?vendor_id=1224)
- [43] Ji, Chuan. (2011, October 19). *How Rooting Works: A Technical Explanation of the Android Rooting Process*. Retrieved from <https://jichu4n.com/posts/how-rooting-works-a-technical-explanation-of-the-android-rooting-process/>
- [44] San-Tsai Sun, Andrea Cuadros, and Konstantin Beznosov. 2015. Android Rooting: Methods, Detection, and Evasion. In Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM '15). Association for Computing Machinery, New York, NY, USA, 3–14.  
DOI:<https://doi.org/10.1145/2808117.2808126>
- [45] Android Developers. *Android Debug Bridge (adb)*. Retrieved from <https://developer.android.com/studio/command-line/adb>
- [46] Zhang Z., Wang Y., Jing J., Wang Q., Lei L. (2014) Once Root Always a Threat: Analyzing the Security Threats of Android Permission System. In: Susilo W., Mu Y. (eds) Information Security and Privacy. ACISP 2014. Lecture Notes in Computer Science, vol 8544. Springer, Cham. Retrieved from [https://link.springer.com/chapter/10.1007/978-3-319-08344-5\\_23](https://link.springer.com/chapter/10.1007/978-3-319-08344-5_23)
- [47] Android Developers. *System and kernel security*. Retrieved from <https://source.android.com/security/overview/kernel-security>
- [48] Android Developers. *Security-Enhanced Linux in Android*. Retrieved from <https://source.android.com/security/selinux>
- [49] Android Developers. *Application security*. Retrieved from <https://source.android.com/security/overview/app-security>
- [50] CCN-CERT IA-13/19. (2019, May). *Ciberamenazas y Tendencias*. Retrieved from <https://www.ccn-cert.cni.es/informes/informes-ccn-cert-publicos/3776-ccn-cert-ia-13-19-ciberamenazas-y-tendencias-edicion-2019-1/file.html>
- [51] Avira Protection Labs. (2020, September 29). *Malware Threat Report: Q2 2020 Statistics and Trends*. Retrieved from <https://www.avira.com/en/blog/malware-threat-report-q2-2020-statistics-and-trends>
- [52] G DATA. (2021, February 7). *G DATA threat report: Number of cyber attacks increases significantly in the first quarter*. Retrieved from <https://www.gdatasoftware.com/blog/2020/07/36199-number-of-cyber-attacks-increases-significantly-in-the-first-quarter>

- [53] Avira Protection Labs. (2021, February 8). *Q4 and 2020 Malware Threat Report*. Retrieved from <https://www.avira.com/en/blog/q4-and-2020-malware-threat-report>
- [54] Chebyshev, Victor. (2021, March 1). SecureList. *Mobile malware evolution 2020*. Retrieved from <https://securelist.com/mobile-malware-evolution-2020/101029/>
- [55] Miguel Ángel García del Moral, Malware en Android: Discovering, Reversing & Forensics, 0xWORD.
- [56] AV-Comparatives. (2019, March 12). Android Test 2019 – 250 Apps. Retrieved from <https://www.av-comparatives.org/tests/android-test-2019-250-apps/>
- [57] W. Yao et al., "Security Apps under the Looking Glass: An Empirical Analysis of Android Security Apps," 2020 IEEE 45th Conference on Local Computer Networks (LCN), 2020, pp. 381-384, doi: 10.1109/LCN48667.2020.9314784. Retrieved from <https://ieeexplore.ieee.org/document/9314784>
- [58] AV-Comparatives. (2021, July 6). *Mobile Security Review 2020*. Retrieved from <https://www.av-comparatives.org/tests/mobile-security-review-2020/#protection-against-android-malware>
- [59] AV-TEST. (2021, March). *The best antivirus software for Android*. Retrieved from <https://www.av-test.org/en/antivirus/mobile-devices/>
- [60] Samani, Raj. (2020, Q1). McAfee. *McAfee Mobile Threat Report*. Retrieved from <https://www.mcafee.com/content/dam/consumer/en-us/docs/2020-Mobile-Threat-Report.pdf>
- [61] Wikipedia. *Android Studio*. Retrieved from [https://en.wikipedia.org/wiki/Android\\_Studio](https://en.wikipedia.org/wiki/Android_Studio)
- [62] AWS. *Amazon EC2*. Retrieved from <https://aws.amazon.com/ec2/?ec2-whats-new.sort-by=item.additionalFields.postDateTime&ec2-whats-new.sort-order=desc>
- [63] *SQLite*. Retrieved from <https://sqlite.org/index.html>
- [64] *GitHub Desktop*. Retrieved from <https://desktop.github.com/>
- [65] Android Developers. *Sign your app*. Retrieved from <https://developer.android.com/studio/publish/app-signing?hl=en#app-signing-google-play>
- [66] Google Play services. *Authenticating Your Client*. Retrieved from <https://developers.google.com/android/guides/client-auth?hl=en>
- [67] Plesky, Elvis. (2018, November 20). Plesk. *Linux Logs Explained*. Retrieved from <https://www.plesk.com/blog/featured/linux-logs-explained/>
- [68] Android Developers. *Logging*. Retrieved from <https://developer.android.com/ndk/reference/group/logging>

- [69] Android Developers. *Android.util.Log*. Retrieved from  
<https://developer.android.com/reference/android/util/Log#public-methods>
- [70] Android Developers. *Logcat*. Retrieved from  
<https://developer.android.com/studio/command-line/logcat>
- [71] Wikipedia. *Log Analysis*. Retrieved from  
[https://en.wikipedia.org/wiki/Log\\_analysis](https://en.wikipedia.org/wiki/Log_analysis)
- [72] Solarwinds Loggly. *Android Logging and Log Viewer Tool*. Retrieved from  
<https://www.loggly.com/solution/android-log-viewer/>
- [73] Lopatkin, Mikhail. Bitbucket. *android-log-viewer: Tool for viewing application logs from Android devices*. Retrieved from  
<https://bitbucket.org/mlopatkin/android-log-viewer/src/master/>
- [74] Android Developers. *Permissions on Android*. Retrieved from  
<https://developer.android.com/guide/topics/permissions/overview>
- [75] Android Developers. *Request app permissions*. Retrieved from  
<https://developer.android.com/training/permissions/requesting#one-time>
- [76] VirusShare. *Hashes*. Retrieved from  
[https://viruss share.com/ hashes](https://virusshare.com/ hashes)
- [77] Android Developers. *PackageInfo*. Retrieved from  
<https://developer.android.com/reference/android/content/pm/PackageInfo>
- [78] Android Developers. *PackageManager*. Retrieved from  
<https://developer.android.com/reference/android/content/pm/PackageManager>
- [79] Android Developers. *Manifest.permission*. Retrieved from  
<https://developer.android.com/reference/android/Manifest.permission>
- [80] Android Permissions. *All you ever wanted to know about Android permissions*. Retrieved from <http://androidpermissions.com/>
- [81] VirusTotal. Retrieved from  
<https://www.virustotal.com/gui/>
- [82] Android Developers. *SDK Platform Tools*. Retrieved from  
<https://developer.android.com/studio/releases/platform-tools>
- [83] VirusShare. *Malware sample MD5 list for VirusShare\_00008.zip*. Retrieved from  
[https://viruss share.com/hashfiles/VirusShare\\_00008.md5](https://viruss share.com/hashfiles/VirusShare_00008.md5)
- [84] Android Developers. *Application Sandbox*. Retrieved from  
<https://source.android.com/security/app-sandbox>
- [85] Shinchao's Notes. *Chapter 4: Elementary TCP Sockets*. Retrieved from  
<https://notes.shichao.io/unp/ch4/>

- [86] Android Developers. *Android AsyncTask*. Retrieved from  
<https://developer.android.com/reference/android/os/AsyncTask>
- [87] Android Developers. *Verified Boot*. Retrieved from  
<https://source.android.com/security/verifiedboot>
- [88] Android Developers. *PackageInfo: Requested Permissions*. Retrieved from  
<https://developer.android.com/reference/android/content/pm/PackageInfo#requestedPermissions>
- [89] Atkinson, Michelle. (2015, November 10). Pew Research Center. *Chapter 3: An Analysis of Android App Permissions*. Retrieved from  
<https://www.pewresearch.org/internet/2015/11/10/an-analysis-of-android-app-permissions/>
- [90] Android Developers. *ApplicationInfo Constants*. Retrieved from  
[https://developer.android.com/reference/android/content/pm/ApplicationInfo#constants\\_1](https://developer.android.com/reference/android/content/pm/ApplicationInfo#constants_1)
- [91] Android Developers. *INSTALL\_PACKAGES*. Retrieved from  
[https://developer.android.com/reference/android/Manifest.permission#INSTALL\\_PACKAGES](https://developer.android.com/reference/android/Manifest.permission#INSTALL_PACKAGES)
- [92] Android Developers. *Use Fingerprint permission*. Retrieved from  
[https://developer.android.com/reference/android/Manifest.permission#USE\\_FINGERPRINT](https://developer.android.com/reference/android/Manifest.permission#USE_FINGERPRINT)
- [93] Google Play. *WhatsApp Messenger*. Retrieved from  
[https://play.google.com/store/apps/details?id=com.whatsapp&hl=en\\_US&gl=US](https://play.google.com/store/apps/details?id=com.whatsapp&hl=en_US&gl=US)
- [94] Google Play. *Instagram*. Retrieved from  
[https://play.google.com/store/apps/details?id=com.instagram.android&hl=en\\_US&gl=US](https://play.google.com/store/apps/details?id=com.instagram.android&hl=en_US&gl=US)
- [95] Google Play. *Diccionario RAE y ASALE (DLE)*. Retrieved from  
[https://play.google.com/store/apps/details?id=es.rae.dle&hl=es\\_419&gl=US](https://play.google.com/store/apps/details?id=es.rae.dle&hl=es_419&gl=US)
- [96] Google Code. *Android Market API*. Retrieved from  
<https://code.google.com/archive/p/android-market-api/>
- [97] Google Play. *TikTok*. Retrieved from  
[https://play.google.com/store/apps/details?id=com.ss.android.ugc.trill&hl=en\\_US&gl=US](https://play.google.com/store/apps/details?id=com.ss.android.ugc.trill&hl=en_US&gl=US)
- [98] Android Developers. *Uninstall Shortcut permission*. Retrieved from  
[https://developer.android.com/reference/android/Manifest.permission#UNINSTALL\\_SHORTCUT](https://developer.android.com/reference/android/Manifest.permission#UNINSTALL_SHORTCUT)
- [99] Github. *Nasip Kismet degilmis*. Retrieved from  
<https://github.com/AndroidWordMalware/malware-samples/blob/master/b32d064261a49b80f78fc3578eaa2cdf7c6a0dc0.apk>
- [100] Github. *SmartcardService*. Retrieved from  
<https://github.com/AndroidWordMalware/malware-samples/blob/master/4419a50191600cc7c09d4314576ad0fc5e4e49bb.apk>
- [101] Github. *Android-Malware-Analyzer*. Retrieved from  
<https://github.com/dapuente13/Android-Malware-Analyzer>