APKAnalyzer—Advanced Instrumentation and Malware Detection Through Dynamic Analysis of Android Applications Using Frida

Universitatea Politehnica Timisoara
Faculty of Automation and Computer Science
Computers and Information Tehnology

Student: **Albert Endre-László**

Scientific coordinators:
drd.ing Alexandru Bozdog
dr.Habil.ing Mihai Udrescu-Milosav

Rezumat

Această lucrare prezintă o aplicație demonstrativă care are scopul de a detecta și semnala dacă o aplicație Android este malițioasă, utilizând atât analiza statică cât și analiza dinamică. Aplicația se bazează pe un toolkit de instrumentare dinamică numit Frida, care permite dezvoltatorilor să intercepteze și să modifice comportamentul aplicațiilor Android în timp real. Analiza statică implică examinarea codului sursă al aplicației fără a-l executa, permițând identificarea unor modele și vulnerabilități cunoscute.

Analizatorul instalează aplicațiile Android în medii izolate, mai exact pe emulatoare, și efectuează o analiză dinamică automată. Emulatoarele replică mediul unui dispozitiv real, permițând rularea aplicațiilor într-un context controlat, fără riscul de a compromite datele reale ale utilizatorului. Aceasta asigură că orice comportament malițios al aplicațiilor este detectat într-un mediu sigur, protejând astfel informațiile sensibile ale utilizatorilor.

Pentru a simula comportamentul uman în timpul instalării și interacțiunii cu aplicația, analizatorul folosește un instrument numit AndroidViewClient. Acesta permite automatizarea interacțiunilor cu interfața utilizatorului, cum ar fi acordarea permisiunilor și navigarea prin meniuri. Prin utilizarea AndroidViewClient, analizatorul poate emula în mod eficient acțiunile pe care un utilizator le-ar întreprinde în mod normal, declanșând astfel comportamentele ascunse ale aplicațiilor malițioase. Această metodă ajută la identificarea comportamentelor suspecte care ar putea fi activate doar în prezența unor anumite condiții de utilizare, oferind o imagine completă asupra potențialului malițios al aplicațiilor analizate.

În plus, este prezentată o aplicație Android, care evidențiază capacitatea analizatorului de a detecta și semnala APK-uri malițioase în timpul proceselor de descărcare sau instalare pe dispozitivul utilizatorului, precum și în urma unei scanări dedicate. Această aplicație demonstrativă efectuează monitorizarea continuă a dispozitivului și, la detectarea unui nou fișier APK, inițiază automat procesul de analiză.

Utilizatorul este informat în timp real despre rezultatele analizei, având posibilitatea de a verifica dacă aplicația respectivă a încercat să execute acțiuni malițioase.

Un aspect esențial al acestei aplicații este utilizarea bazelor de date pentru stocarea și gestionarea informațiilor colectate în timpul analizelor. Baza de date MongoDB a fost aleasă pentru a stoca rezultatele analizei dinamice și statice, datorită capacității sale de a oferi interogări rapide. Pentru partea de front-end, aplicația Android utilizează biblioteca Room pentru gestionarea bazei de date SQLite locale, care păstrează jurnalele analizei și istoricul scanărilor efectuate pe dispozitiv.

Pentru a asigura acuratețea și eficiența analizatorului, au fost efectuate teste riguroase și o validare detaliată a performanțelor. Testarea a implicat utilizarea unui set variat de fișiere APK, incluzând atât aplicații benigne, cât și aplicații cunoscute pentru comportamentele lor malițioase. Fiecare aplicație a fost supusă atât analizei statice, cât și celei dinamice, pentru a evalua capacitatea analizatorului de a detecta diferite tipuri de amenințări.

De asemenea, testele de performanță au fost esențiale pentru a determina impactul utilizării aplicației Android asupra resurselor sistemului al utilizatorului. Aceste teste au confirmat că aplicația poate rula eficient pe dispozitivele utiliziatorilor consumând foarte puține resurse.

Contents

List of Figures

Listings

List of Tables

1 Introduction

Over the past 15 years, smart mobile devices have become increasingly important in the lives of people, and now they are inseparable. With its many functions, from the use of computers to the payment of goods in stores and online, it is not surprising that these devices generally contain a lot of information about their users. In environments abundant with information, there's an inherent desire for some to appropriate it or to use it illicitly. This is done by running computer programs on the target device, and tries to steal information or perform certain malicious actions on the target device. These programs are called malware.

1.1 Abstract

This paper presents a proof-of-concept application that aims to detect and flag if an Android application is malicious using both static and dynamic analysis. The application is based on a dynamic instrumentation toolkit Frida. The application installs Android applications in isolated environments, specifically emulators, and conducts automatic dynamic analysis. The analyzer uses a tool called AndroidViewClient to simulate human behaviour when installing and interacting with the application.

Additionally, a proof-of-concept Android application is presented to demonstrate the analyzer's ability to flag malicious APKs during download or installation on a user's phone, or as a result of a scan.

1.2 Research motives

Malware behavior is largely dependent on the underlying operating system. For example, malware targeting Windows must be written specifically for that OS. While frameworks like Flutter facilitate cross-platform application development for both Android and iOS, these are outside the scope of this study. The current market is dominated by two mobile operating systems: Android and iOS, with variations like MIUI (based on Android) used in specific devices. StatCounter is a web traffic analysis website which provides statistics from over 3 million websites based on StatCounters official documentation.[1]

To ensure digital security, it is crucial to identify whether an application is malicious. This can be achieved through two main methods: static analysis and dynamic analysis. Static analysis examines the program's code without executing it, often requiring decompiled or disassembled code in the absence of source code. Dynamic analysis, on the other hand, involves running the program in an isolated environment and observing its behavior. Typically, this is done using virtual machines or emulators in the case of Android. Dynamic analysis can include various procedures, from monitoring file system changes to debugging or instrumenting the running process.

After having determined the most common mobile operating system based on figure 1.1, I decided to write an application analyzer for android to determine and flag if an application is malicious using both static and dynamic analysis. The analyzer aims to use both static and dynamic analysis to achieve optimal results. For dynamic analysis I have decided to use a dynamic instrumentation toolkit called Frida. In order to achieve the best possible results I also wrote a module that attempts to emulate human behaviour when installing and interacting

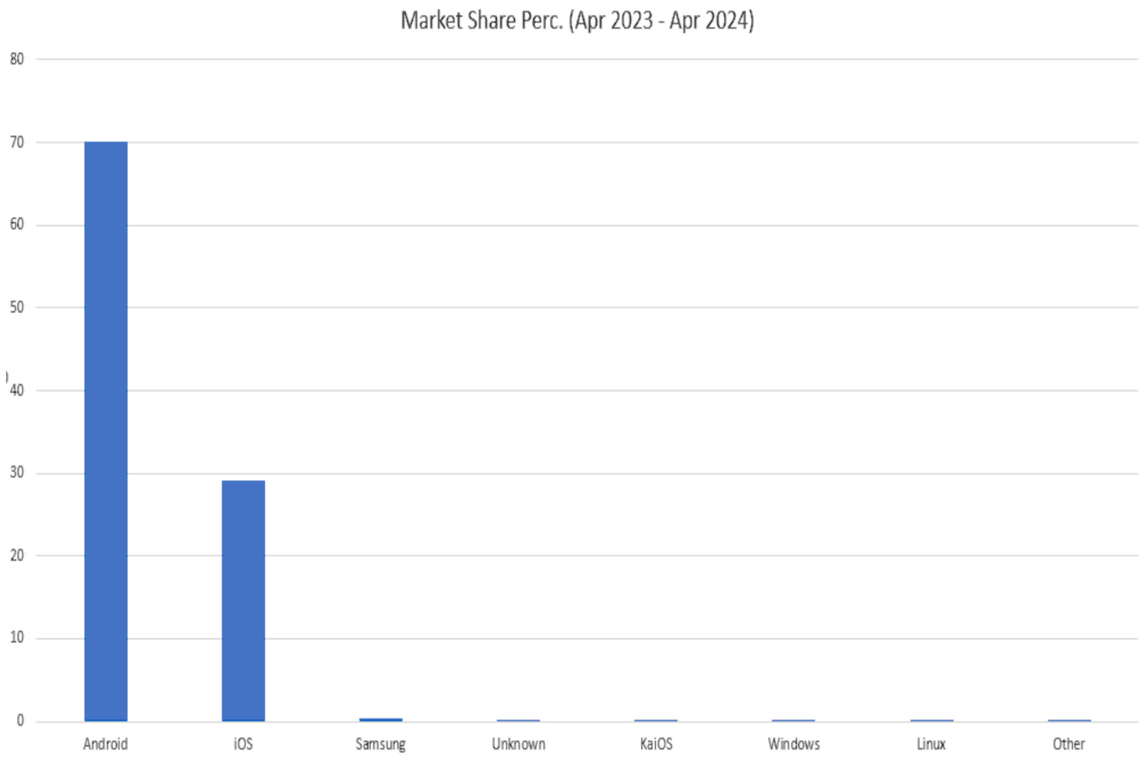


Figure 1.1: Graphic that shows the market distribution of mobile operating systems. On the x axis we have the operating systems, on the y axis their respective percentage.

with the application with the help of AndroidViewClient. Along with the analyzer I also set out to write a proof-of-concept Android application that showcases the analyzers' capabilities and how it can be used to protect users from malicious applications.

The analyzer in its essence is a web server. Upon receiving a request to start an analysis, the analyzer selects an emulator that is available from a pool of emulators and sets up the Frida environment on the given emulator. The application instruments the received APK file with certain Frida hooks, after which the analysis is started. A script I have created using AndroidViewClient will simulate user interaction with the emulator. The client can continuously query the server for the status of the analysis and results of the analysis are updated real time.

The Android application I have created to demonstrate the capabilities of the analyzer attempts to monitor the device on which it is installed. When an APK file is copied to the storage of the device, the application sends the APK file to be analysed. The user can see the outcome of the emulation in real time and see if the application attempted to do anything malicious. Figure 1.2 presents a general overview of the application.

The choice of using Frida for dynamic analysis and the implementation of simulated user interactions via AndroidViewClient represent significant advancements in my approach to malware detection. These methods allow for a more thorough examination of APK behavior in a controlled environment, capturing a wide range of potential malicious activities. This project not only addresses current threats but also establishes a foundation for ongoing improvements and adaptations to emerging malware techniques, ensuring its relevance and effectiveness in the continually evolving landscape of mobile security.

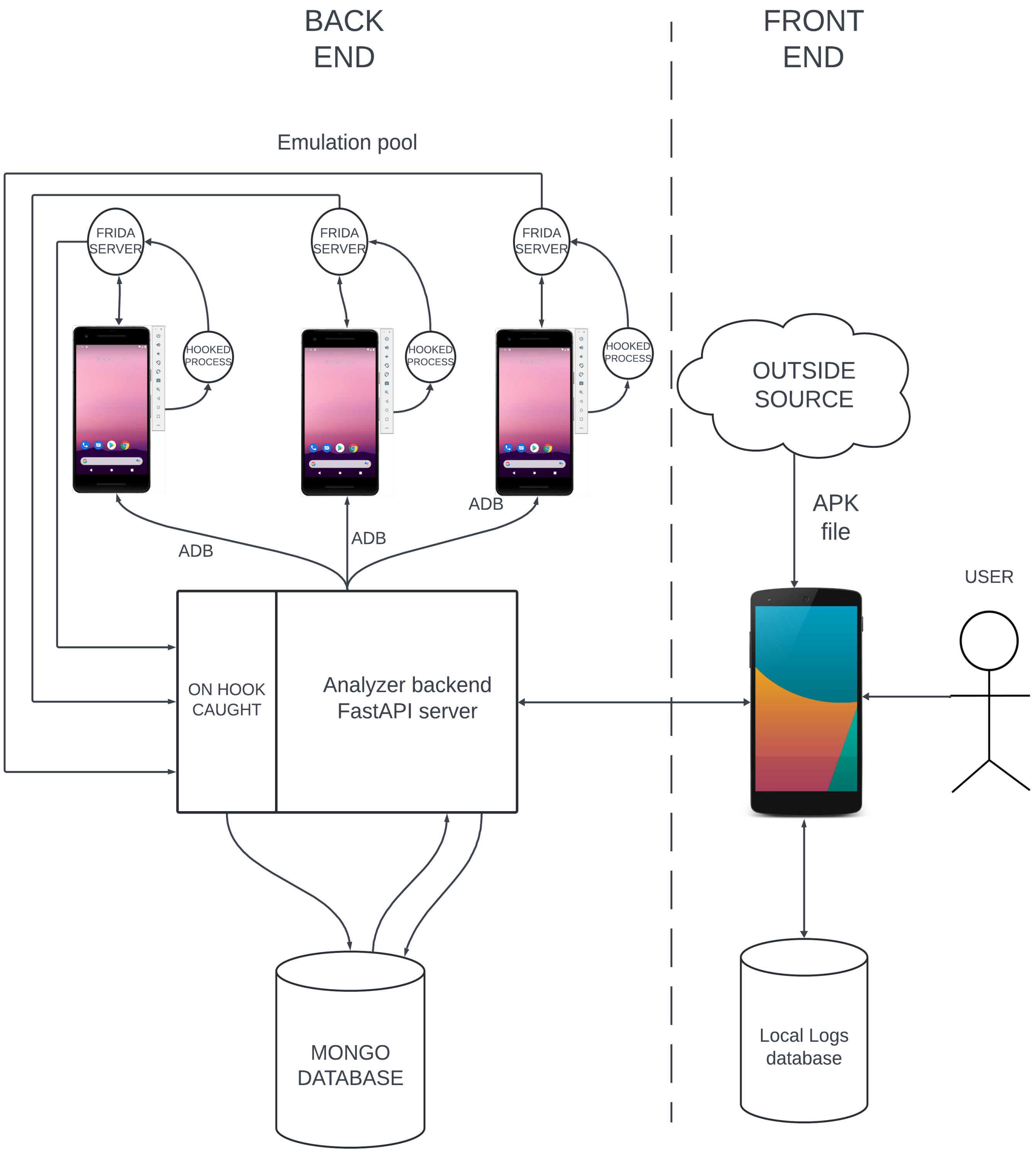


Figure 1.2: General overview of the application. Left side of the dotted line is the back-end, right side is the user side Android application.

1.3 Paper structure

This section intends to provide the reader with an outline of the paper.

In the initial section, I offer a broad overview of the topic and explain several reasons why this type of application proved beneficial for malware detection. This part also discusses the rationale behind my decision to develop malware detection tools specifically for Android as opposed to other platforms or operating systems. Additionally, I introduce several key concepts essential for comprehending the project's scope and nature.

The second part aims to give readers an overview of the state of the art technologies and the current state of mobile antivirus. Popular technologies (hooking frameworks) and techniques used for both viruses and anti-viruses are presented [2]. These are aimed at keeping readers up-to-date with current Android malware and how it works in the Android ecosystem, as well as some common technologies for detecting and indicating malicious applications. Similar applications that use hooking techniques are also briefly presented and compared to my own implementation.

The third section focuses on the design aspects of the project. Here I describe the chosen programming languages (Python, Java, JavaScript), what frameworks and third party libraries I have used throughout the application (Frida, AndroidViewClient, Virustotal) but also the storage mechanisms used throughout (MongoDB, local SQLite database on the user-side). I also explain certain design choices I made.

The fourth section focuses on demonstrating the implementation of both server-side and client-side applications. It covers various elements of code design and general development considerations. The section includes code examples and images from both applications.

In Chapter 5, I will explain the testing process of the application. This part will cover the errors encountered and the solutions implemented during development, as well as the testing procedures and the outcomes obtained. It includes representative images, code snippets, and figures.

Chapter 6 provides a summary of the initial objectives and the final product. It also discusses potential future enhancements for the application.

2 State of the Art

The field of Android malware is characterized by rapid evolution, with new threats emerging frequently and antivirus solutions continuously developing novel strategies to counteract these threats. Understanding the current state of Android malware and the methodologies used to combat it is essential for grasping the scope and functionality of the application developed in this project. Additionally, analyzing similar applications available in the market provides valuable insights into their strengths and weaknesses, allowing for a comprehensive assessment of our application's capabilities. This comparative analysis is crucial for identifying potential gaps and tailoring future development efforts to address these deficiencies, ensuring the application remains effective and relevant in the dynamic landscape of Android malware.

2.1 Modern Malware

Modern malware has evolved significantly, becoming more sophisticated and dangerous, targeting various platforms and exploiting multiple vulnerabilities. This section examines three significant malware families: Anatsa, Xenomorph, and SharkBot, highlighting the advanced techniques employed by contemporary threats. I will also mention a fourth family of less malicious malware called Adware. Modern malware often uses a combination of social engineering, sophisticated coding techniques, and advanced evasion strategies to compromise devices and steal sensitive information. The rapid evolution of these threats requires continuous updates to security measures and constant vigilance from users and cybersecurity professionals.

The landscape of modern malware is diverse, encompassing a wide range of attack vectors and methodologies. These malware families represent a fraction of the numerous threats that evolve daily, leveraging advanced tactics such as automated transfer systems, overlay attacks, and covert distribution methods to achieve their malicious objectives. By understanding the characteristics and behaviors of these malware families, we can better appreciate the complexity and severity of the current cybersecurity threat environment.

2.1.1 Xenomorph: A Resilient Threat

Xenomorph, discovered in early 2022, is a sophisticated Android malware known for its resilience and adaptability. Targeting a wide array of financial institutions and cryptocurrency wallets, Xenomorph employs overlay attacks to mimic legitimate banking apps, tricking users into entering their credentials. Similar to Anatsa, Xenomorph utilizes an Automated Transfer System (ATS) engine to facilitate fraudulent transactions and supports a variety of commands. These commands include simulating touches and preventing devices from sleeping, which enhances its control over infected devices. Xenomorph is often distributed through phishing campaigns masquerading as legitimate software updates, leading to numerous infections across multiple regions.

Xenomorph's adaptability is further demonstrated by its ability to evade detection and persist on infected devices. Its overlay attacks are particularly deceptive, as they present fake login screens that are almost indistinguishable from the genuine apps, thereby effectively capturing users' credentials. The malware's use of ATS enables it to perform transactions directly from the compromised device without alerting the user, thereby facilitating substantial financial theft.

The distribution methods of Xenomorph are primarily through phishing campaigns. These campaigns often impersonate software updates or other legitimate notifications, thereby deceiving users into downloading the malware. Once installed, Xenomorph can remain undetected for extended periods, continuously harvesting sensitive information and executing fraudulent transactions. [3]

2.1.2 Anatsa: Android Banking Malware

Anatsa is a potent Android banking malware known for its complex capabilities and widespread impact. It primarily spreads through malicious apps available on legitimate platforms like Google Play. Anatsa employs advanced techniques such as automated transfer systems (ATS), allowing it to automate fraudulent transactions, and exploits accessibility services to gain extensive control over infected devices. The malware features stealth capabilities, enabling it to hide its presence by deleting its icon and avoiding detection by antivirus software. The malware is designed with sophisticated evasion techniques. It can remove its own icon from the home screen, making it difficult for users to identify and remove the malicious app. Anatsa also employs methods to avoid detection by antivirus software, including delaying its malicious activities to bypass behavioral analysis during the initial installation period.

Recent campaigns have demonstrated Anatsa's effectiveness in targeting users through phishing attacks, which are often disguised as legitimate notifications or app updates. These attacks lead to significant financial theft by capturing banking credentials and other sensitive information. The widespread impact of Anatsa is evident as it has managed to infect a large number of devices, causing substantial financial losses to individuals and financial institutions alike. [4]

2.1.3 SharkBot: Covert Distribution via Droppers

SharkBot is an Android banking trojan that employs covert distribution methods to evade detection. Disguised as legitimate apps on Google Play, SharkBot droppers request permissions like REQUEST_INSTALL_PACKAGES to install malicious payloads. The malware uses anti-emulator checks and targets users by verifying if the SIM ISO corresponds with specific countries like Italy and Great Britain. SharkBot then checks if users have installed targeted banking applications. Despite being removed from Google Play, these droppers persist on third-party stores, posing ongoing threats. I contributed to the analysis of SharkBot, providing insights into its distribution methods and impact. [5]

2.1.4 Adware: Persistent and Annoying

While not as immediately dangerous as banking malware, adware remains a significant threat due to its pervasive and intrusive nature. Adware infiltrates devices through seemingly harmless applications and bombards users with unwanted advertisements. These ads can appear as pop-ups, notifications, or within applications, significantly degrading user experience and device performance. Some adware variants also collect user data, track browsing habits, and redirect users to malicious websites, posing additional privacy and security risks. The persistence of adware on the Google Play Store highlights the challenges of maintaining a secure app ecosystem and the need for continuous monitoring and user vigilance [6].

2.2 Similar Implementations

In light of the sophisticated techniques employed by modern malware such as Anatsa, Xenomorph, and SharkBot, it's essential to examine existing security analysis applications and their approaches to combating these threats. Understanding the capabilities and limitations of similar implementations can provide valuable insights into how current technologies address the challenges posed by advanced malware. This section delves into several notable security analysis applications, evaluating their methodologies and effectiveness in identifying and mitigating malware threats. By comparing these tools to our application, we can highlight the unique strengths and innovations our approach brings to the table. Understanding the strengths and weaknesses of these tools can also provide valuable insights into the most effective strategies for detecting and mitigating modern malware. This comparative analysis is essential for identifying potential gaps and tailoring future development efforts to address these deficiencies, ensuring the application remains effective and relevant in the dynamic landscape of Android malware.

2.2.1 Mobile Security Framework (MobSF)

Mobile Security Framework (MobSF) is an automated, all-in-one mobile application (Android/iOS/Windows) pen-testing, malware analysis, and security assessment framework. It is capable of performing both static and dynamic analysis. MobSF includes a dynamic analyzer that supports both Android and iOS applications, offering a platform for interactive instrumented testing, runtime data, and network traffic analysis. One of MobSF's strengths is its multi-platform support and extensive static analysis capabilities. MobSF allows users to upload applications via a web interface for static or dynamic analysis. The dynamic analysis is performed on a user-side emulator rather than remotely. However, at the time of writing, I was unable to get the framework's dynamic analysis component to function, so my observations are based on the documentation and source code provided. MobSF is actively maintained and available at `https://github.com/MobSF/Mobile-Security-Framework-MobSF/tree/master`.

2.2.2 AndroPyTool

AndroPyTool is another tool for extracting static and dynamic features from Android APKs. It integrates several well-known Android app analysis tools such as DroidBox, FlowDroid, Strace, AndroGuard, and VirusTotal. Given a directory containing APK files, AndroPyTool applies these tools to perform pre-static, static, and dynamic analysis, generating feature files in JSON and CSV formats, and saving the data in a MongoDB database. The tool's implementation is detailed in the following article [7]. It is important to note that for its dynamic analysis, AndroPyTool relies on DroidBox, a tool that has not been updated for nine years. The DroidBox repository is found at [8]. Additionally, the dynamic analysis performed by DroidBox involves patching the Android kernel. This approach means addition of new hooks would require patching and restarting all the emulators.

2.3 Comparison

After thoroughly examining the available solutions in the market, it becomes crucial to compare these findings to evaluate the unique value our analyzer offers. To facilitate this comparison, Table 2.1 has been constructed. This table provides a detailed comparison of the features present in similar applications alongside those of our application. By juxtaposing these attributes, we can clearly illustrate the distinctive advantages and innovations our analyzer brings to the table.

Features	MobSF	AndroPy	ApkAnalyzer
Static analysis	✓	✓	✓
Dynamic Analysis	✓	✓	✓
User interface	✓	✗	✓
Stimulation of Analysis	✗	✗	✓
Live results	✗	✗	✓
Anti VM features	✓	✗	✗
Multi Platform support	✓	✗	✗
Server side dynamic analysis	✗	✗	✓
Local database for logs	✗	✗	✓
Server side database	✓	✗	✓
Third Party Software integration	✓	✓	✓

Table 2.1: Table that compares two other Android file analyzer applications, namely MobSF and AndroPy with my own application - ApkAnalyzer. Collumns feature the applications, and the rows represent each feature. Features present in an application are marked with a checkmark, while those absent are marked with an X.

The comparative analysis of MobSF, AndroPyTool, and ApkAnalyzer reveals several key insights into their capabilities and limitations. MobSF excels in providing comprehensive static and dynamic analysis across multiple platforms (Android, iOS, and Windows). It also offers a user-friendly interface, making it accessible to a wide range of users. However, its dynamic analysis is limited to user-side emulation, which may restrict its effectiveness in some scenarios.

AndroPyTool, on the other hand, integrates multiple well-known analysis tools to perform detailed static and dynamic analysis. Despite this, it relies on outdated technologies like DroidBox for dynamic analysis, which diminishes its reliability and effectiveness in detecting modern threats. Additionally, the lack of a user interface makes it less accessible for non-technical users.

ApkAnalyzer addresses some of these limitations by providing server-side dynamic analysis and stimulating real-time user interactions, which enhances its ability to detect and analyze malicious behavior accurately. The inclusion of live results and a user interface further improves its usability and accessibility. However, ApkAnalyzer does not support multi-platform analysis and lacks some advanced features like anti-VM techniques found in MobSF. Anti-VM features are easy to add as they can be implemented with the help of Frida hooks.

A notable feature of ApkAnalyzer, which will be elaborated on in the design section, is its highly modular architecture that facilitates seamless extension of dynamic analysis capabilities. This modularity ensures that new analysis methods and tools can be integrated with minimal effort, allowing the system to evolve and adapt to emerging threats and techniques in Android malware. Furthermore, ApkAnalyzer leverages Frida for dynamic analysis, a powerful instrumentation toolkit that handles most of the dynamic analysis tasks through hooking. This approach significantly simplifies the analysis process by centralizing the dynamic analysis operations within Frida, thereby reducing the dependency on additional components and streamlining the overall workflow. This design not only enhances the flexibility and scalability of ApkAnalyzer but also ensures a more efficient and effective analysis of Android applications.

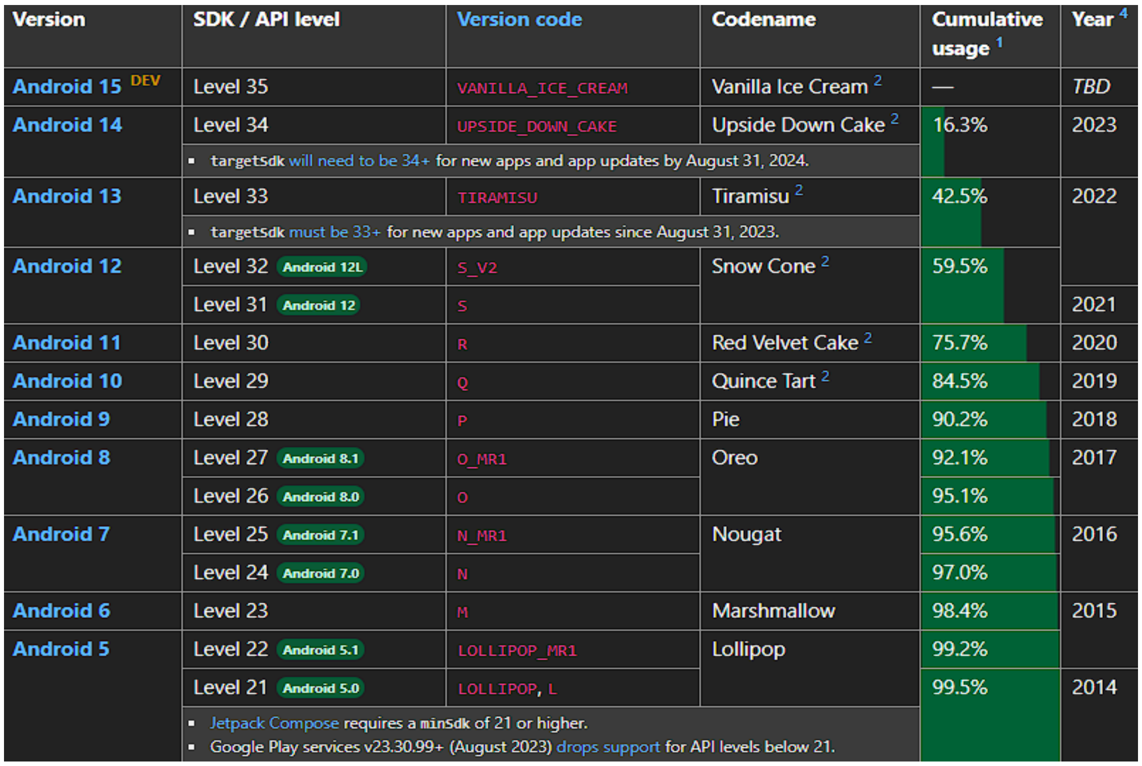
By comparing these tools, it is evident that while MobSF and AndroPyTool offer substantial capabilities, they have limitations in dynamic analysis and user accessibility, respectively. ApkAnalyzer, with its modular architecture and use of Frida, presents a more adaptable and efficient approach.

3 Architectural framework

The domain of Android malware is swiftly advancing, marked by the continual appearance of new obstacles. This dynamic nature fosters a perpetual cat-and-mouse game between malware creators and antivirus developers. Once a new type of malware is developed, antivirus products respond by implementing detection mechanisms. In turn, malware creators modify their software to evade these defenses, prompting further advancements in antivirus technologies [9]. To grasp the intricacies of my application, it is essential to delve into certain theoretical aspects of the Android ecosystem and the nature of malware targeting this platform.

3.1 Android Platform

Android is an open source operating system developed by Google, based on the Linux kernel, and designed for a wide array of devices. Each version of Android is identified by a codename, version number, version code, and SDK/API level. These identifiers, while sometimes used interchangeably, serve specific purposes. The website `apilevels.com` provides a comprehensive overview of all Android versions and their corresponding identifiers. Figure 3.1 presents a detailed summary of Android versions.



Version	SDK / API level	Version code	Codename	Cumulative usage [1]	Year [4]
Android 15 DEV	Level 35	`VANILLA_ICE_CREAM`	Vanilla Ice Cream [2]	—	*TBD*
Android 14	Level 34	`UPSIDE_DOWN_CAKE`	Upside Down Cake [2]	16.3%	2023
	▪ `targetSdk` will need to be 34+ for new apps and app updates by August 31, 2024.				
Android 13	Level 33	`TIRAMISU`	Tiramisu [2]	42.5%	2022
	▪ `targetSdk` must be 33+ for new apps and app updates since August 31, 2023.				
Android 12	Level 32 Android 12L	`S_V2`	Snow Cone [2]	59.5%	
	Level 31 Android 12	`S`			2021
Android 11	Level 30	`R`	Red Velvet Cake [2]	75.7%	2020
Android 10	Level 29	`Q`	Quince Tart [2]	84.5%	2019
Android 9	Level 28	`P`	Pie	90.2%	2018
Android 8	Level 27 Android 8.1	`O_MR1`	Oreo	92.1%	2017
	Level 26 Android 8.0	`O`		95.1%	
Android 7	Level 25 Android 7.1	`N_MR1`	Nougat	95.6%	2016
	Level 24 Android 7.0	`N`		97.0%	
Android 6	Level 23	`M`	Marshmallow	98.4%	2015
Android 5	Level 22 Android 5.1	`LOLLIPOP_MR1`	Lollipop	99.2%	
	Level 21 Android 5.0	`LOLLIPOP`, `L`		99.5%	2014
	▪ Jetpack Compose requires a `minSdk` of 21 or higher. ▪ Google Play services v23.30.99+ (August 2023) drops support for API levels below 21.				

Figure 3.1: Table that presents Android versions from version 4 onwards, and their corresponding identifiers as of May 2024.

Android is built on the Linux kernel, where drivers for hardware components such as the display, Bluetooth, and WiFi are located. The Hardware Abstraction Layer (HAL) provides standard interfaces for utilizing these hardware capabilities.

Starting from version 5, all Android apps run in their own processes, each with an instance of the Android Runtime (ART). ART is designed to manage multiple virtual machines running DEX (Dalvik Executable) files. The DEX bytecode format is optimized for minimal memory usage. Java and Kotlin code is compiled into DEX files, which are then executed by ART.

In addition to the Android Runtime (ART), developers can utilize native C/C++ libraries to interact with the Hardware Abstraction Layer (HAL). The use of native code on Android offers performance advantages over Java code, as highlighted in [10]. Although the Java Native Interface (JNI) overhead is relatively low, typically ranging from 1 to 10 microseconds [11], it is generally recommended to minimize the dependence on JNI. Ideally, applications should maintain distinct native and Java layers, with minimal transfer between the two to optimize performance and maintainability.

A critical consideration in the context of malware is the complexity involved in analyzing and debugging native code compared to Java or Kotlin code. This complexity often makes native code an attractive target for malware authors, who embed malicious content in the native layer to evade detection and analysis [12]. The obfuscation of malicious activities within native code poses a significant challenge to security researchers and antivirus solutions.

All these capabilities are available to developers through the Java API framework, which serves as the foundation for Android application development. The framework provides a comprehensive set of APIs that allow developers to leverage the full potential of the Android platform. In essence, all Android applications are written in Java or translated into Java. Kotlin, a language officially supported by Google for Android development, compiles directly to Java bytecode. Even cross-platform development frameworks such as Flutter and React Native incorporate a Java layer, enabling their proprietary engines to seamlessly interface with the Android operating system.

On top of these layers, we eventually reach the Android applications, which are bundled as APK files. These files are essentially ZIP archives that contain several key components necessary for the application to function. Within an APK file, there is a DEX file that contains the compiled bytecode for the Android Runtime (ART) to interpret and execute. Additionally, APK files include native libraries, which are shared objects that contain compiled C / C++ code that interact directly with the hardware and the HAL.

An Android Manifest file is also part of the APK, providing essential information about the application to the Android system, such as the app's package name, components, like activities and services, permissions, and hardware requirements. In addition, various assets, including images, sounds, and other media files used by the application during runtime, are bundled into the APK.

This architecture is illustrated in Figure 3.2, which is adapted from the official Android documentation. This structure ensures that all necessary elements are bundled together, allowing the application to be installed and executed on Android devices efficiently.

This information is taken from the official Android documentation page. [13]

APK files are installed primarily from the Google Play Store, which is the main source for users to download applications, including potential malware. However, applications can also be installed from various other sources on the Internet or through ADB.

The Android Debug Bridge (ADB) is a command-line tool that facilitates communication between a computer and a connected Android device. Essentially, ADB provides a Unix shell for users on the device, reflecting Android's Linux-based nature. ADB operates as a client-server program comprising a client, a server, and a daemon. The daemon runs in the background on

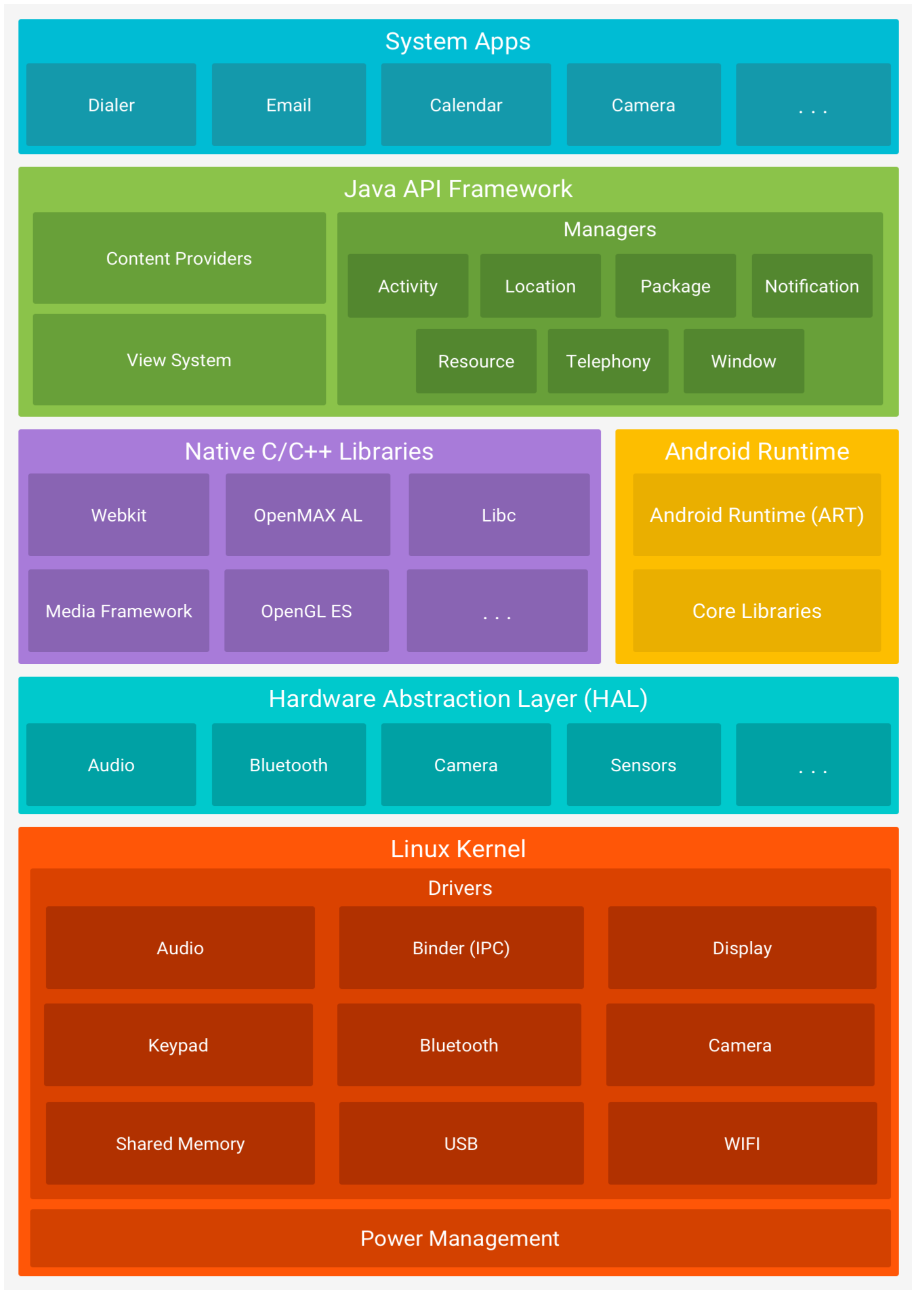


Figure 3.2: Image showing the Android software stack, as previously presented, containing the following: Linux kernel, HAL, ART, Native libraries, Java API framework, applications.

each connected device and is responsible for executing commands. ADB is a crucial tool for development and debugging, and it plays a significant role in multiple aspects of my project.

3.2 Permissions, Manifest, Activities, Services, and Receivers

Every Android application (APK file) must include a manifest file that provides essential information about the application to the operating system. The manifest defines critical details such as the application's package name, its components (activities, services, and receivers), permissions, required hardware and software features, and API levels. Permissions on the Android OS are particularly significant in the context of Android malware.

3.2.1 Permissions and Manifest

For an application to perform operations on sensitive user data or utilize certain system features, it must declare the necessary permissions in the manifest file and request them at runtime. Prior to Android 23, permissions were granted at installation time. A primary objective of malware is to acquire the permissions needed to carry out malicious actions, making the declaration and management of permissions a crucial aspect of Android application security.

The following code snippet in 3.1 illustrate the permission needed for an application to send SMS messages by itself and how it is declared in the manifest file.

Listing 3.1: Android permission in Manifest file.

```
<manifest ... >
    <uses-permission android:name="android.permission.SEND_SMS"/>
    ...
</manifest>
```

At runtime, the application must request this permission from the user. Listing 3.2 illustrates this.

Listing 3.2: Requesting Android permissions in Java.

```
if (ContextCompat.checkSelfPermission(this, Manifest.permission.SEND_SMS) !=
    PackageManager.PERMISSION_GRANTED) {
    ActivityCompat.requestPermissions(this, new
        String[]{Manifest.permission.SEND_SMS}, REQUEST_SMS_PERMISSION);
}
```

There are many permissions like this that can be classified as dangerous. Listing 3.3 presents a non exhaustive list of permissions considered dangerous by Android. An exhaustive list of all permissions can be found on the official Android documentation site [13]:

Listing 3.3: Non exhaustive list of Android permissions.

```
    READ_CALENDAR, WRITE_CALENDAR, CAMERA, READ_CONTACTS, WRITE_CONTACTS,
        RECORD_AUDIO, READ_PHONE_NUMBERS, CALL_PHONE, ANSWER_PHONE_CALLS, SEND_SMS,
        RECEIVE_SMS, READ_SMS
```

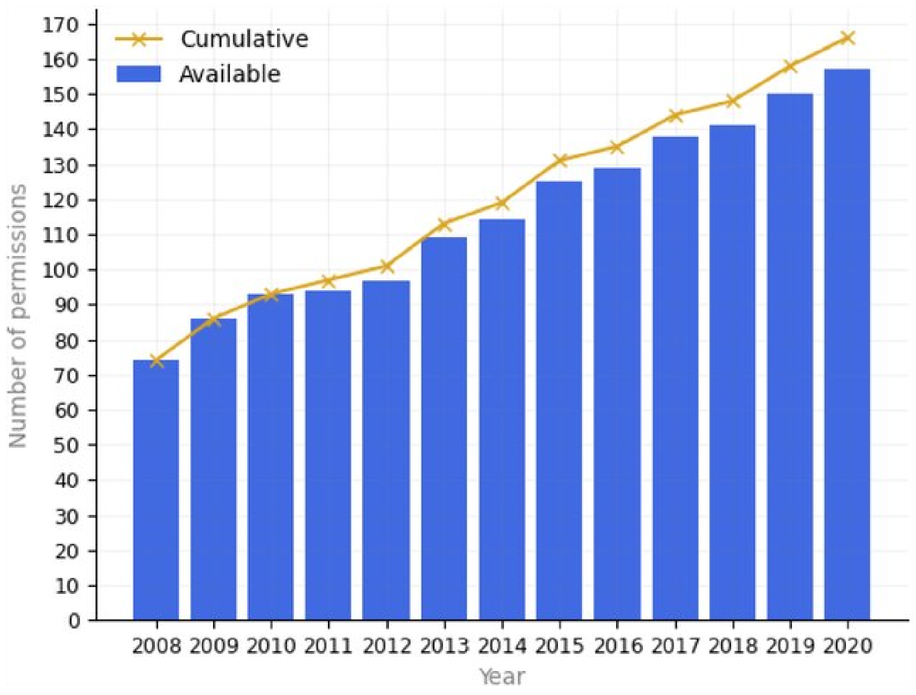


Figure 3.3: Graphic showing the number of available permissions as offered by the Android operating system, over time. Number of permissions on the y axis, time on the x axis. [14]

The names of these permissions are typically self-explanatory. For example, the CAMERA permission allows an application to use the device's camera to take pictures. The misuse of permissions is a primary vector for malicious activities within the Android ecosystem. To mitigate this risk, Google has implemented stricter permission controls and user consent mechanisms in recent versions of Android, ilustrated by figure 3.3. Users are advised to carefully scrutinize permission requests and grant only those permissions that are essential for the application's functionality. Permissions play a critical role in the context of Android malware. Malware often seeks to exploit permissions to perform malicious actions, such as stealing sensitive data, tracking user activities, or sending unauthorized messages, as detailed in [14]. This underscores the importance of robust permission management and vigilant user consent.

3.2.2 Accessibility and device admin

Two specific categories of permissions——accessibility and device administration——merit special attention due to their potential misuse by malware.

Accessibility services are originally intended to assist users with disabilities by offering alternative methods to interact with their devices. However, these capabilities can be exploited by malicious applications to gain significant control over the device. Malware can utilize accessibility services to monitor keystrokes, track app usage, fill out forms automatically, or create overlays to capture sensitive information such as credit card details.

Device administration permissions grant applications control over crucial device policies and settings, which can be exceptionally powerful in the wrong hands. Malware equipped with device admin rights can lock the device, perform a factory reset, or even prevent its own removal, thus maintaining persistence on the device and evading uninstallation attempts. This makes device admin permissions a valuable tool for malware seeking to assert control and remain undetected on compromised devices.

3.2.3 Activities, Services, Receivers

An activity represents a single screen with a user interface, and each activity must be declared within the application element of the manifest file. The main activity, typically designated as the entry point of the application, includes an intent filter specifying it as the main launcher activity. It is important to note that developers can extend the Application class and implement a custom onCreate function, which can also serve as an entry point for applications, including potentially malicious ones.

A service is a component designed to perform operations in the background without a user interface. Malware often utilizes services to ensure persistence, such as maintaining continuous internet connections or executing tasks discreetly in the background.

Broadcast receivers respond to system-wide broadcast announcements by invoking callback functions. These components are commonly leveraged by malware developers to intercept system events. For example, a malicious application could implement a broadcast receiver to capture incoming SMS messages and extract their contents, demonstrating how these receivers can be exploited to perform unauthorized actions.

3.3 Malware techniques

To develop an effective antivirus solution or an APK analyzer, it is crucial to first understand the operational mechanics of malware and the sophisticated techniques it employs to avoid detection. This includes gaining insight into the various strategies used by malware authors, such as obfuscation, encryption, and the exploitation of system vulnerabilities. A thorough understanding of these tactics is essential to identify potential weaknesses in existing security measures and to design robust detection and mitigation strategies.

3.3.1 Types of Malware

According to a 2021 paper on malware [15], there are 8 main categories of Android malware, namely: Adware, Backdoor, Scareware, PUA, File Infector, Riskware, Ransomware, Trojan.

Adware: Adware is designed to display intrusive advertisements on a user's device, particularly during web browsing. It often gathers personal information such as phone numbers, email addresses, application accounts, device IMEI number, device ID, and status. Some sophisticated adware can even use the device's camera to capture images, attempt to encrypt data, and install other malicious programs or files.

Backdoor: Backdoor malware creates hidden access points into a smartphone, allowing attackers to bypass authentication and gain elevated privileges. This type of malware can capture personal information, send and receive messages, make calls, record call history, and gather lists of installed and running programs. In severe cases, backdoors can root the Android device, granting attackers extensive control. Backdoors are often installed through adware, which entices users to click on malicious advertisements.

Scareware: Scareware employs fear tactics to trick users into downloading or purchasing harmful applications. It might, for example, prompt users to install fake software that claims to protect their device. Scareware typically attempts to gather device information, including GPS location, and installs additional malicious malware.

Potentially Unwanted Applications (PUAs): PUAs are applications that, while not inherently harmful, can be packaged with legitimate software. They can collect personal information and user contacts, track the device's GPS location, and display unwanted pop-up advertisements, notifications, warnings, URLs, and shortcuts.

File Infector: This type of malware infects APK files, which contain all data for an application. When these infected APK files are installed, the malware activates. File infectors can slow down devices, drain battery power, collect device ID, IMEI number, and phone status, and potentially gain root access to alter or delete files and applications.

Riskware: Riskware refers to legitimate software that poses security risks due to vulnerabilities. Although the software itself is not malicious, it can collect data, send and receive SMSs, steal network information, connect to malicious websites, install harmful content, and display malicious advertisements.

Ransomware: Ransomware encrypts files and folders on a device, preventing user access until a ransom is paid. It typically involves sending and receiving SMSs, locking SIM cards and phones, collecting Wi-Fi connection data, and interacting with a remote server to manage the ransomware attack.

Trojan: Trojans masquerade as legitimate programs while secretly performing malicious activities. They can delete, alter, block, and copy data, impair operating system services, and steal information without the user's knowledge. Trojans often remain undetected in the background, posing significant security risks. A well built Trojan would be a banker. A well-known banker malware familiy would the Xenomorph. According to an article written by ThreatFabric [3], Xenomorph uses overlays as its main way to obtain Personally Identifiable Information (PII) such as usernames, passwords, credit card numbers, and much more. The control server transmits to the bot a list of URLs containing the address from which the malware can retrieve the overlays for the infected device.

This particular application is meant to be geared towards Trojans, Ransomware, however, since different types of malware often employ similar techniques at the code level, it would be of no surprise if other types of malware would be detected.

3.3.2 General approaches to malware development

There are some common practices when it comes to Android malware. They mainly revolve around the idea of making the program as difficult to decipher by malware analysts.

One widely spread technique would be code obfuscation. This technique relies on making the source code difficult to interpret or comprehend, even if decompiled or disassembled. This table 3.1 is a modified version of a table found in 2021 article [16] about obfuscation on Android.

Another popular method would be dynamic code execution. As its name suggests, dynamic code execution refers to a technique during which an application loads code that is not part of its codebase and executes it. Code loading can be done at the method level using reflection, a mechanism provided by Java, or by loading addition DEX files at runtime. An example can be seen at 3.4.

Listing 3.4: Dynamic code execution in Kotlin. Example taken from https://erev0s.com/blog/3-ways-for-dynamic-code-loading-in-android/.

```
fun cl(dexFileName: String): DexClassLoader {
    val dexFile: File = File.createTempFile("pref", ".dex")
    var inStr: ByteArrayInputStream =
        ByteArrayInputStream(baseContext.assets.open(dexFileName).readBytes())
    inStr.use { input ->
        dexFile.outputStream().use { output ->
            input.copyTo(output)
        }
    }
    var loader: DexClassLoader = DexClassLoader(
        dexFile.absolutePath,
        null,
        null,
        this.javaClass.classLoader
    )
    return loader
}

...

var loader = cl(dexfilename) // get the DexClassLoader
val loadClass = loader.loadClass("com.erev0s.randomnumber.RandomNumber") // get the
    class
val checkMethod = loadClass.getMethod("getRandomNumber") // get the method
val cl_in = loadClass.newInstance() // instantiate the class
checkMethod.invoke(cl_in) as String // invoke the method
```

These DEX files are ususally stored in an encrypted form and later are decrypted and loaded into memory. This complicates static analysis since the exact target is unknown. With the help of Frida however this can be done easily.

Obfuscation technique	**Difficulty of implementation**
APK aligning	low
Manifest transformation	low
Renaming	
Identifier renaming	low
Package renaming	low
Native library stripping	low
Control flow modification	
Code shrinking	low
Call indirection	low
Junk code insertion	low
Code reordering	low
Reflection	high
Evasion attacks	high
Encryption	
Data encryption	low
Asset file encryption	mid
Class encryption	high
Native code encryption	high

Table 3.1: Overview of obfuscation techniques. Left collumn shows obfuscation technique, right column shows implementation difficulty.

3.3.3 Anti Debugging Techniques

To circumvent dynamic analysis, malware writers rely on Anti-Debugging techniques.

One of the primary methods of anti-debugging involves detecting the presence of debugging tools on the device. This can be done by checking for specific system properties or files that indicate a debugger is attached. For example, applications can look for the android.os.Debug.isDebuggerConnected() method, which returns true if a debugger is connected to the application. Additionally, the presence of certain files or processes commonly used by debuggers, such as /system/bin/gdbserver, can also be checked.

Applications frequently check if a device is rooted, as this is often indicative of a sandbox environment. Sophisticated malware also searches for the presence of hooking applications like Frida or Xposed, which are commonly used for dynamic analysis. The cat-and-mouse aspect mentioned in the beginning [9] begins to show itself here.

3.4 Malware Analysis techniques

Understanding and countering Android malware requires employing a variety of analysis techniques. These techniques are broadly classified into static and dynamic analysis, each with its distinct methods and objectives.

3.4.1 Static Analysis

Static analysis involves examining the application's code and resources without executing it. This method aims to identify malicious patterns, vulnerabilities, and potentially harmful behaviors embedded in the application's codebase. Tools like JADX and APKTool are used to convert the compiled bytecode into a human-readable format. In order to automatically extract information from an APK one can also use Androguard. Androguard has a friendly Python API.

3.4.2 Dynamic Analysis

Dynamic analysis involves executing the application in a controlled environment to observe its behavior in real-time. This method helps uncover hidden malicious activities that may not be evident through static analysis alone. Dynamic analysis is typically conducted within a sandbox or virtualized environment. In the case of Android this is done using Android emulators. During execution, the application's behavior is closely monitored. Analysts look for activities such as network communications, file system modifications, and interactions with other applications. Tools like Frida and Xposed allow analysts to hook into the application's runtime and intercept function calls. This technique provides deep insights into the application's behavior and can be used to manipulate its execution flow for further analysis.

3.5 Back-end Technologies

This section aims to cover the technologies used in the back-end of my application. In order to create a back-end that is viable and has adequate performance, I had to choose the technology stack carefully. After a long and careful analysis these are the technologies that I chose to utilize.

3.5.1 Python

The primary programming language for the back-end is Python. Python is a high-level, general-purpose language known for its simplicity and readability. Several features make Python particularly suitable for this project, such as garbage collection and strong support for object-oriented programming. Additionally, many essential tools and frameworks, such as Frida, Androguard, and AndroidViewClient, either exclusively support Python or have the most intuitive and user-friendly Python APIs.

3.5.2 FastAPI

For the web framework, I selected FastAPI. According to its official documentation [17], FastAPI is a modern, fast (high-performance) web framework for building APIs with Python, based on standard Python type hints [17]. FastAPI serves as the backbone of the server, efficiently handling all incoming requests and ensuring high performance and scalability. In this paper [18], the authors suggest that FastAPI performs well when it comes to CPU and memory usage, which is crucial when we are operating with Frida and several emulators simultaneously.

3.5.3 Androguard

Androguard is a Python-based tool designed to analyze and parse APK files. It plays a crucial role in the static analysis phase, extracting essential information from APK files, such as the permissions required, package name, and application label. Additionally, Androguard retrieves the target SDK version, which is instrumental in selecting an appropriate emulator for dynamic analysis—a process that will be elaborated on in the section discussing emulators. The official Androguard Documenation [19] proved to be a good starting point when it came to learning how to use the tool.

3.5.4 AndroidViewClient

AndroidViewClient is a Python tool utilized for automated testing on Android devices and emulators. It provides a programmatic approach to simulating user inputs on a device. While tools of this nature are typically employed for testing purposes, I used AndroidViewClient to simulate user interactions to trigger as many malware functionalities as possible. For instance, I programmed the emulator to automatically accept runtime permissions, thereby enabling a thorough analysis of the malware's behavior in a controlled environment.

3.5.5 Frida

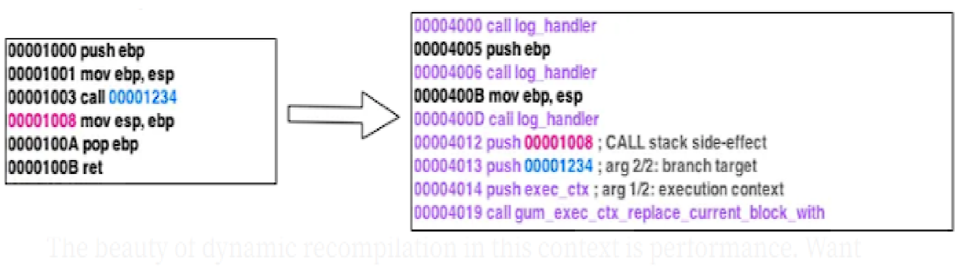
The hooking framework of choice for this project is Frida. Frida is a powerful dynamic code instrumentation toolkit that allows the injection of JavaScript snippets into native applications on multiple platforms, with a primary focus on Android.

At its core, Frida is written in C, and it utilizes the QuickJs JavaScript engine to execute JavaScript code. Frida injects this engine into the target app's process, enabling it to run with full access to the app's memory. This setup is where Frida excels: it allows you to hook into functions or invoke other native functions within the process. Additionally, Frida establishes a communication channel between the target app's process and the hosting process (in this case, the back-end), facilitating seamless data exchange and control.

To utilize Frida's capabilities, root access is required, because Frida needs to write to other processes' memory. This requirement is easily met by using emulators that can be configured to provide the necessary root access. Frida's ability to dynamically hook into and manipulate app processes makes it an invaluable tool for detailed and effective malware analysis.

Another significant advantage of Frida is its ease of use. Unlike other hooking frameworks that may require modifying emulator images or patching the kernel with specific hooks, Frida simplifies the process significantly. All that is needed is to push the necessary Frida executable onto the target device and start it. This straightforward setup process improves the accessibility and usability of Frida for dynamic code instrumentation and malware analysis on Android devices. A great article [20] presents the numerous advantages provided by Frida in comparison to other function hooking tools. A notable downside of Frida is that it is more resource-intensive than other hooking applications.

Stalker is the core component of Frida. According to the official Frida documentation [21], Stalker is Frida's code tracing engine, capable of following threads and capturing every function, block, and instruction executed. It is compatible with the AArch64 architecture for mobile devices (Android/iOS) and Intel 64 and IA-32 architectures for desktops and laptops.



```
00001000 push ebp
00001001 mov ebp, esp
00001003 call 00001234
00001008 mov esp, ebp
0000100A pop ebp
0000100B ret
```

```
00004000 call log_handler
00004005 push ebp
00004006 call log_handler
0000400B mov ebp, esp
0000400D call log_handler
00004012 push 00001008 ; CALL stack side-effect
00004013 push 00001234 ; arg 2/2: branch target
00004014 push exec_ctx ; arg 1/2: execution context
00004019 call gum_exec_ctx_replace_current_block_with
```

The beauty of dynamic recompilation in this context is performance. Want

Figure 3.4: Illustration that shows how Stalker code tracer operates. Left side shows the uninstrumented section, right side shows code with intstrumentation.

The creator of Frida, who also developed the code tracing engine, authored an excellent article that introduces the concept behind Stalker [22]. [22].

The concept is straightforward. When a thread is about to execute its next instructions, you duplicate those instructions and embed logging code into the duplicate. In this way, the original instructions remain unchanged, preserving anti-debug checksum integrity, while the copied version is executed. For example, you would insert a CALL to the log handler before each instruction, providing details about the upcoming execution [22]. This is done with the help of the unix call **ptrace**.

The advantage of dynamic recompilation in this context lies in its performance benefits. If tracing every single instruction is required, logging code can be inserted between each copied instruction. If the focus is solely on CALL instructions, logging code can be added exclusively alongside those specific instructions. Consequently, runtime overhead is introduced only as necessary, based on the specific tracing requirements [22].

The figure shown at 3.4 illustrates this concept very well.

Despite all this, it is important to keep in mind that Frida is not completely undetectable from a malware standpoint. Malware often tries to detect if it is being analyzed, and if it finds traces of ongoing analysis, it will alter its behavior. In [23], the authors discuss some mechanisms used to detect Frida [23]. Ideally, analysis tools should implement mechanisms to hide their presence.

3.5.6 MongoDB

MongoDB is a widely used document-oriented database program. It falls under the category of NoSQL databases, designed for high performance, scalability, and flexibility in managing unstructured data. Unlike traditional relational databases, MongoDB stores data in JSON-like documents, making it easier to represent complex hierarchical relationships and schemas.

In the context of this project, MongoDB serves as a robust and scalable backend storage solution for storing and querying data generated during the analysis of Android applications. Its ability to handle large volumes of data and provide fast queries makes it well-suited for storing the results of dynamic and static analyses performed on APK files.

3.5.7 VirusTotal bindings

VirusTotal is an established online service widely recognized for its comprehensive analysis of suspicious files and URLs. Leveraging VirusTotal within my application provided a robust mechanism to scrutinize URLs requested by target applications, thereby enabling the detection of potential interactions with malicious websites.

VirusTotal aggregates results from multiple antivirus engines and various scan modes, offering a holistic assessment of the safety and reputation of files and URLs. Integrating VirusTotal into the application workflow involved utilizing its API bindings, which facilitated the submission of URLs from the target application for real-time analysis.

3.5.8 Emulators and ADB

Emulators replicate the functionalities of physical Android devices, offering advantages such as flexibility, reproducibility, and ease of configuration. For this project, emulators play a pivotal role in creating controlled testing environments where malware behaviors can be observed without risking real devices or compromising user data. By leveraging emulators, the project ensures a standardized testing environment across different scenarios and device configurations.

ADB is a versatile command-line tool that facilitates communication between a host computer and an Android device or emulator. It enables various operations such as installing and debugging applications, accessing the device's file system, and pulling information from the device. In this project, ADB is instrumental in managing and interacting with emulators deployed for dynamic analysis.

3.6 Front-end Technologies

The proof-of-concept Front-end I have created to demonstrate the capabilities of the analyzer is essentially an Android application.

3.6.1 Java

Java serves as the primary programming language for the front-end implementation. Its selection was driven by its intuitive nature and robust capabilities, particularly in monitoring the device's file system and facilitating seamless communication with the back end.

Java's object-oriented programming paradigm provided a structured approach to developing the front-end, ensuring clarity in code organization and maintenance. Leveraging Java's extensive libraries and frameworks specific to Android development, such as Android SDK and Android Studio, streamlined the implementation of essential functionalities.

The use of Java in the front-end allowed for efficient handling of user interactions, UI components, and data processing tasks required for interacting with the back end. Its compatibility with Android's ecosystem ensured optimal performance and responsiveness, crucial for delivering a smooth user experience during malware analysis and threat detection.

3.6.2 Room database

Another essential technology utilized in the Android application is the Room library, an integral component provided by Android for managing local SQLite databases. Room serves as the backbone for storing logs pertaining to the analysis of APK files.

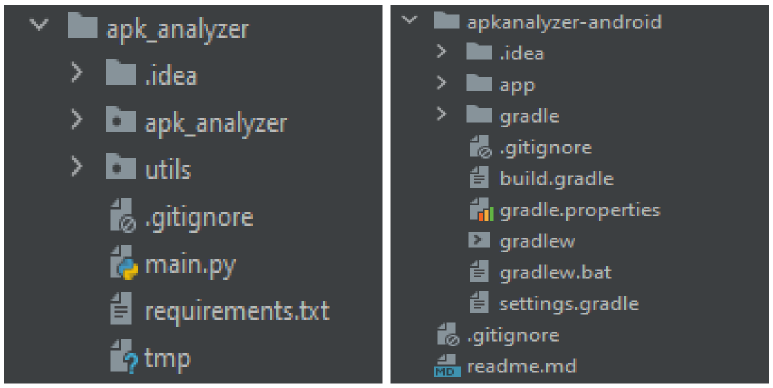
Room database is a robust, object-relational mapping (ORM) library that simplifies the integration and management of SQLite databases within Android applications. It offers a layer of abstraction over SQLite, enabling developers to interact with databases using high-level Java or Kotlin objects.

The local database enables users to track analysis history.

3.7 Application structure

At the root of the application we have the 2 important folders: **apk_analyzer** and **apkanalyzer-android**. The first one represents the back-end of the application written in Python while the second one represents the so-called front-end written in Java.

Figure 3.5 presents the structure of the back-end and the front-end of the application.



```
apk_analyzer
  .idea
  apk_analyzer
  utils
  .gitignore
  main.py
  requirements.txt
  tmp
```

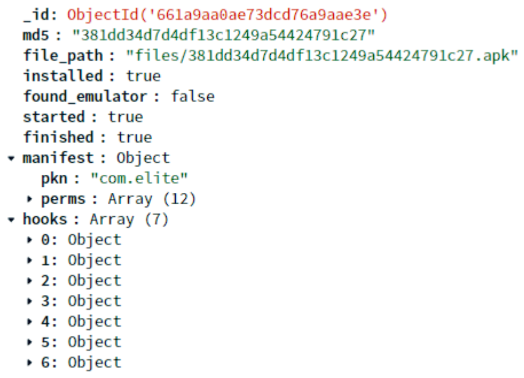
```
apkanalyzer-android
  .idea
  app
  gradle
  .gitignore
  build.gradle
  gradle.properties
  gradlew
  gradlew.bat
  settings.gradle
.gitignore
readme.md
```

Figure 3.5: Back-end and Front-end folder structure of Apkanalyzer.

3.8 Database structure

In my project, I utilized MongoDB as the database solution due to its flexibility and scalability in handling large volumes of unstructured data. MongoDB's document-oriented structure is particularly suited for the complex and nested data generated during the analysis of Android APK files. I chose to maintain a single database and collection for simplicity and efficiency, ensuring that all related data points are centralized and easily accessible. This approach facilitates streamlined queries and management of data, which is crucial for analyzing patterns and behaviors across multiple APKs.

Each entry in the database represents a comprehensive record of the analysis performed on a specific APK file. For instance, an entry includes fields such as the file's MD5 hash, file path, installation status, and detailed information from the APK's manifest. Additionally, it captures dynamic analysis results, including hooks on various methods, timestamps, and other relevant details. This structure ensures that all pertinent information is stored cohesively, allowing for efficient retrieval and analysis. The decision to use MongoDB and maintain a single collection underscores the importance of a robust and easily navigable data storage solution in supporting the backend functionalities of the application. Figure 3.6 illustrates how I stored data in the collection.



```
  _id: ObjectId('661a9aa0ae73dcd76a9aae3e')
  md5 : "381dd34d7d4df13c1249a54424791c27"
  file_path : "files/381dd34d7d4df13c1249a54424791c27.apk"
  installed : true
  found_emulator : false
  started : true
  finished : true
▾ manifest : Object
    pkn : "com.elite"
  ▸ perms : Array (12)
▾ hooks : Array (7)
  ▸ 0: Object
  ▸ 1: Object
  ▸ 2: Object
  ▸ 3: Object
  ▸ 4: Object
  ▸ 5: Object
  ▸ 6: Object
```

Figure 3.6: Example of document stored in mongoDB visualised on an online platform.

3.9 Code versioning

GitHub is a widely used platform for hosting and collaborating on software development projects. It serves as a distributed version control system (VCS) based on Git, allowing developers to manage and track changes to their codebase over time. Git projects are stored in repositories where each repository also contains the history of modifications.

For my project, I chose to use GitHub due to its robust version control capabilities and ease of use. GitHub's features, such as commit tracking and branch management, made it simple to organize and manage the development process. The platform's ability to document and revert changes ensured that I could maintain a clear and accurate record of the project's evolution, making development more efficient and manageable. Figure 3.7 presents the commit history of the application.

The repository of the application can be found at https://github.com/albert020119/apkanalyzer.

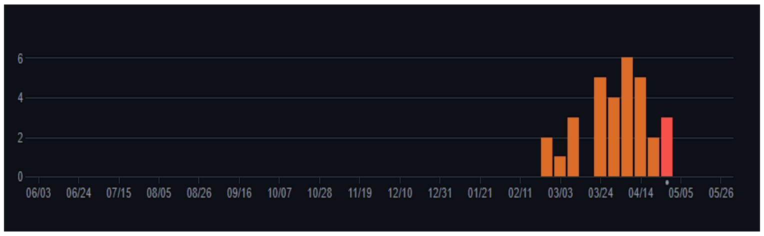


Figure 3.7: Commit history of the application, with number of commits on the y axis and time on the x axis.

In summary, this theoretical overview has covered essential aspects of the Android ecosystem, malware behaviors, and the technologies and methodologies used in the development of my application. Understanding these foundational elements is crucial for appreciating the design and functionality of the application. The next section will delve into the implementation details, providing a comprehensive look at how these concepts and technologies were applied to build the system.

4 Development and integration

The implementation of this project involves the integration of various technologies and tools to create a robust system for analyzing Android APK files. This section will delve into the specifics of how the back-end and front-end components were developed, the methodologies employed, and the rationale behind the chosen technology stack. The goal is to provide a detailed walk-through of the implementation process, highlighting key decisions and the functionality of the application.

The back-end of the system is designed to perform both static and dynamic analysis of APK files. Python was selected as the primary programming language due to its extensive library support and compatibility with essential tools such as Frida, Androguard, and Android-ViewClient. FastAPI serves as the web framework, facilitating efficient request handling and API management. Data storage is managed using MongoDB, ensuring scalable and flexible data handling capabilities.

On the front-end, an Android application was developed to demonstrate the system's capabilities. Java was chosen for its robust support for Android development, allowing seamless monitoring of the device's file system and communication with the back-end. The Room database library is utilized for local storage of analysis logs.

The general flow of the application operates as follows: When a user has my application installed, any new APK file appearing on their device is automatically sent to the backend for analysis. During the analysis process, the user-side application continuously pings the server to receive updates on the status of the analysis.

The backend initiates the analysis by first extracting relevant information from the APK using Androguard. The APK then installs on an emulator, loads the necessary hooks, and begins dynamic analysis. As hooks are triggered, their results are written to the database, ensuring that the user-side application receives the latest updates during its periodic pings. Once the analysis is completed and the finished flag is set, the user-side app is notified that the analysis has concluded.

In the subsequent sections, each component of the implementation will be explored in detail, starting with the back-end technologies, followed by the front-end development, and concluding with the integration and testing processes. This comprehensive examination will illustrate how the system works cohesively to provide a thorough analysis of Android APK files.

4.1 Back End

The backend implementation is a critical component of my application, responsible for handling the core functionality of analyzing APK files. The following subsections provide a detailed overview of the backend architecture.

4.1.1 FastAPI

A crucial component of the application is the web server, implemented using FastAPI. FastAPI is a modern, high-performance web framework for building APIs with Python, known for its ease of use and speed. In my implementation, the server provides two primary endpoints that facilitate the interaction between the user-side application and the backend analysis system.

The first endpoint is designed to receive an APK file and initiate its analysis in the background. This non-blocking approach ensures that the user receives an immediate response, thus enhancing the user experience by not requiring them to wait for the entire analysis to complete. Upon receiving the file, the server generates an MD5 checksum to uniquely identify the APK, which is then returned to the user. This MD5 checksum serves as a reference for all subsequent interactions and status checks related to the file.

The second endpoint is the status endpoint, which handles continuous pings from the user-side application. This endpoint takes the MD5 checksum as an argument and performs a query in the database to retrieve the current state of the analysis. It then aggregates the results and returns an AnalysisStatus object, which encapsulates the current status and progress of the analysis. This allows the user-side application to provide real-time updates to the user about the ongoing analysis. The implementation of these endpoints is illustrated in the code snippet at 4.1, showcasing how FastAPI's capabilities are leveraged to manage the asynchronous analysis workflow efficiently.

Listing 4.1: Handlers of the server-side; one for starting the analysis the other for retrieving information about ongoing analysis.

```python
@router.post("/apk", response_model=StartAnalysis)
async def analyze_apk(file: UploadFile, background_tasks: BackgroundTasks):
    downloaded_file_path = download_file(file)
    checksum = calc_md5(downloaded_file_path)
    analyzer.analyzer_db.add_new(checksum, downloaded_file_path)
    background_tasks.add_task(analyzer.analyze, downloaded_file_path, checksum)
    return StartAnalysis(md5=checksum, filename=file.filename)


@router.get("/status", response_model=Union[AnalysisStatus, SampleNotFound])
async def update_status(md5: str):
    result = await analyzer.get_status(md5)
    if not result:
        return SampleNotFound()
    return result
```

Code snippet 4.2 presents how the server is actually started.

Listing 4.2: Code associated with initializing the FastAPI server on localhost.

```python
import uvicorn as uvicorn

from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
from apk_analyzer import handlers


def build_app():
    app = FastAPI()
    app.add_middleware(
        CORSMiddleware,
        allow_origins=['http://localhost:3000'],
        allow_credentials=True,
        allow_methods=["*"],
        allow_headers=["*"],
    )

    app.include_router(handlers.router)
    return app


if __name__ == "__main__":
    application = build_app()
    uvicorn.run(application, host="0.0.0.0", port=8000)
```

The implementation leverages FastAPI's asynchronous capabilities to handle tasks in the background, thereby improving response times and user experience. Additionally, the server setup includes essential middleware configurations and endpoint handlers, demonstrating a robust and scalable approach to managing API requests.

4.1.2 Emulator and Emulator Pool

A critical component of the application is the use of emulators for installing and running applications. To facilitate this, I developed an Emulator class that models the behavior and management of an emulator. The class is outlined in code snippet 4.3.

Listing 4.3: Blueprint of Emulator class with its corresponding methods.

```
class Emulator:
    def __init__(self, emulator_path: str, avd: str, port: str):
        self.emulator_path = emulator_path
        self.avd = avd
        self.adb = ADB(ADBConfig, port=port)
        self.viewclient = None
        self.device = None
        self.serialno = None
        self.port = port
        self.available = True
        self.frida_session = None
        self.clown = None

    def start_emulator(self):
        ...

    @staticmethod
    def run_command(executable, *args):
        ...

    def install_sample(self, path):
        ...
    def uninstall(self, pkn: str):
        ...
```

Listing 4.4 shows the actual implementation of some of the methods seen in listing 4.3.

Listing 4.4: Implementation of run command, install sample and uninstall methods.

```
    @staticmethod
    def run_command(executable, *args):
        command = [executable]
        command.extend(args)
        subprocess.Popen(command, shell=True, close_fds=True, stdout=None,
            start_new_session=True)

    def install_sample(self, path):
        output, _ = self.adb.cmd(["install", path])
        return output.decode('utf-8')

    def uninstall(self, pkn: str):
        output, _ = self.adb.cmd(['uninstall', pkn])
        return output.decode('utf-8')
```

One notable feature of the Emulator class is its thread-like behavior. Each emulator can be either busy or free. When an analysis task is assigned to an emulator, it is 'locked' to prevent concurrent analyses on the same device. Once the analysis is complete, the emulator is 'unlocked,' making it available for new tasks. This mechanism ensures that multiple applications are not analyzed simultaneously on a single emulator, maintaining the integrity and reliability of the analysis process. This effect is achieved with the **lock** and **release** methods.

An emulator can be selected using the **EmulationPool**. This class functions as a manager for the collection of available emulators, ensuring efficient allocation and utilization. It provides an interface for other components of the analyzer to request an available emulator—specifically one that is not currently locked for analysis tasks. Additionally, the EmulationPool class supports operations that need to be executed across all emulators, facilitating coordinated actions. The structure and methods of this class are visible at 4.5.

Listing 4.5: Blueprint of EmulationPool class, with it's methods shown.

```
class EmulationPool:
    def __init__(self, emulation_config: EmulatorConfig):
        ...

    def get_available_emulator(self, apk) -> Emulator | None:
        ...

    def start_emulators(self):
        ...

    def setup_frida_all(self):
        ...
```

The Emulator and EmulationPool classes form a critical part of the application's infrastructure, ensuring efficient management and utilization of emulators for application analysis. The Emulator class provides a comprehensive model for interacting with individual emulators, while the EmulationPool class manages the allocation and coordination of these resources.

4.1.3 ADB

To facilitate communication between the server and the emulators, I chose to use the Android Debug Bridge (ADB). ADB provides a Unix shell on each device, allowing all commands and emulator setups to be executed efficiently. To streamline this communication and enhance code design, I encapsulated the ADB functionality within a dedicated class. The code snippet at 4.6 shows the method used to execute a command using ADB.

Listing 4.6: cmd method of ADB class, used to run commands on an emulator.

```
def cmd(self, commands: list):
    command = [self.adb_path, '-s', 'emulator-' + self.port]
    command.extend(commands)
    process = subprocess.run(command, shell=True, close_fds=True,
        capture_output=True)
    return process.stdout, process.stderr
```

4.1.4 Setup

When starting the server, a comprehensive setup process is performed on each device to ensure proper configuration of Frida. This process includes identifying and downloading the latest version of the Frida server compatible with the emulator's architecture. Once the appropriate Frida server is obtained, it is installed on the emulator. Finally, the Frida server is started on each device, ensuring that the environment is correctly prepared for subsequent analysis tasks. This setup is crucial for enabling instrumentation and dynamic analysis capabilities provided by Frida. The method shown at 4.7 is called on each device when the server is started.

Listing 4.7: Setup frida method of the Emulator class, called on each emulator on initial setup.

```
def setup_frida(self):
    version, cpu_arch = get_frida_latest(self.adb)
    download_frida_server(version=version, cpu_arch=cpu_arch)
    install_frida_server(self.adb, cpu_arch=cpu_arch)
```

4.1.5 Hooks

Hooks constitute the main Hooks form the core part of the application. Data is gathered using Frida hooks, which are implemented in JavaScript following Frida's guidelines. These hooks are found in the **frida.hooks** package shown in 4.1, and they are all loaded into the Frida server on the emulator just before the target application is launched.

Currently, I have implemented hooks of these types: extbfaccessibility, dex, library, network, telephony. Due to the modular design, they can be easily extended. Information extracted

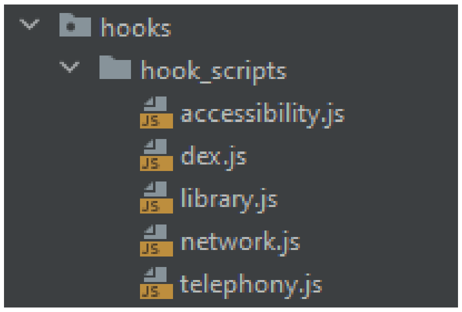


Figure 4.1: Folder structure of the JavaScript files which contain the Frida hooks

by the hooks varies depending on their specific purpose. To standardize and manage this information, I modeled it using the **HookEvent**, a dataclass that encapsulates the details extracted by each hook. This dataclass, detailed in 4.8, provides a structured way to handle the various types of data generated during the analysis process, ensuring consistency and ease of use across the application.

Listing 4.8: Class that models a hook event, along with its attributes and possible types.

```
@dataclass
class HookEvent:
    type: str
    method: str
    code: int
    timestamp: int
    args: list | str | None
    return_value: str | None
```

Code snippet 4.9 demonstrates how a Frida hook is implemented to intercept the **SmsManager.sendTextMessage** method. This hook specifically targets the overload of the method that includes parameters for destination address, service center address, text message content, and two pending intents for sending and delivery. When the method is called, the hook captures crucial details such as the destination address and message text. It then constructs a **hookEventData** object containing the method type, name, a unique code, timestamp, and the arguments passed to the method. This object is serialized into a JSON string and sent to the Frida server for logging or further processing before the original **sendTextMessage** method is executed. The code present in the message is a rating of the severity for the caught event. In this context, the severity code ranges from 1 to 3, with 1 indicating a negligible event and 3 indicating a severe event, suggesting that the application might be malicious. This approach allows for real-time monitoring and analysis of SMS-related activities within the application. The format I have created ensures I can pass both return values and parameters of functions.

Listing 4.9: JavaScript implementation of a Frida hook, intended to override the sendTextMessage method of the SmsManager class, and forward information to the python server.

```
SmsManager.sendTextMessage.overload('java.lang.String', 'java.lang.String',
    'java.lang.String', 'android.app.PendingIntent',
    'android.app.PendingIntent').implementation = function(destAddr, scAddr, text,
    sentIntent, deliveryIntent) {
        var timestamp = new Date().getTime();
        var hookEventData = {
            type: 'telephony',
            method: 'SmsManager.sendTextMessage',
            code: 3,
            timestamp: timestamp,
            args: [destAddr, text],
            return_value: null
        };
        send(JSON.stringify(hookEventData));
        return this.sendTextMessage.apply(this, arguments);
    };
```

Hooks are received in Python and managed by the **HookHandler**. This handling is facilitated by a function called in a callback style. When a hook event is triggered, the HookHandler aggregates the data received from the JavaScript hooks and writes it to a MongoDB database. For network-related events, the associated URLs are further scrutinized for potential threats. These URLs are scanned using the VirusTotal API, leveraging their official service to check for any malicious content or behavior.

4.1.6 Jester

To enhance the functionality of the analyzer, I implemented a solution to simulate user interactions, aiming to achieve more accurate results. For example, if an application lacks the **SEND_SMS** permission, it may not attempt to send SMS messages, rendering the corresponding hook ineffective. To address this issue, I developed the **Jester** module, which automates user input within the application. This module can autonomously grant permissions requested by the application, including complex ones like Accessibility or Device Admin permissions. If no permissions are requested, the Jester module simulates user actions by clicking random visible buttons on the screen. This functionality leverages the **AndroidViewClient**, originally designed for creating automated UI tests on Android applications. Communication between the Jester module and the emulator is facilitated through ADB, ensuring seamless interaction and accurate simulation of user behaviors. To give example of it's functionality, the method at 4.10 attempts to determine if the current view contains a permission prompt.

Listing 4.10: A method of the Jester class used to determine if the currently visible view contains a permission prompt. If a permission prompt is indeed present it will return the ID of the Allow button.

```
    @staticmethod
    def identify_permission_screen(dump: list[View]) -> View | None:
        buttons = [view for view in dump if view['class'] == 'android.widget.Button']
        allow_button = [view for view in buttons if
                        view['resource-id'] == 'permission_allow_button']
        decline_button = [view for view in buttons if
                          view['resource-id'] == 'permission_deny_button']
        if allow_button and decline_button:
            return allow_button[0]
        return None
```

The simulation starts by initializing the Jester class with an emulator, a view client, the APK under test, and a runtime duration. The execution begins with the start method, which records the start time to manage the runtime duration. The main loop continues to execute, checking via the end_loop method whether the specified runtime has elapsed.

Every 0.5 seconds, the algorithm dumps the current view hierarchy using the view client to obtain the current screen state. It first looks for standard permission prompts using the identify_permission_screen method. If found, it touches the ”Allow” button and sends a random SMS to simulate interaction. It also identifies special permission screens, such as those for Accessibility or Device Admin, using the identify_special_permission_screen method, and if found, it touches the relevant button to grant the permission.

In conclusion, the Jester module significantly enhances the capabilities of the analyzer by providing a robust framework for simulating user interactions and handling permission requests autonomously. This advanced level of automation ensures that even complex permission scenarios are addressed seamlessly, improving the accuracy and efficiency of the testing process. By leveraging tools such as the AndroidViewClient and ADB, Jester offers a unique and comprehensive solution that sets it apart in the industry, enabling developers to thoroughly test their applications under realistic conditions with minimal manual intervention. This not only streamlines the development cycle but also helps in identifying potential issues early, leading to more reliable and user-friendly applications.

4.1.7 MongoDB

Data storage utilizes MongoDB, with the server establishing a connection to the database upon startup. Each instance of the **HookHandler** is equipped with a database handler to facilitate efficient data management. As previously described, the database operates with a single collection where detailed information from each hook event is systematically stored and managed. An entry in the collection is modeled by the dataclasses shown at 4.11 and the code associated with initializing the connection can be seen at 4.12.

Listing 4.11: SampleEntry and ManifestInfo classes designed to model a database entry.

```
@dataclass
class ManifestInfo:
    pkn: str
    label: str
    permissions: List[str]
    ...

@dataclass
class SampleEntry:
    md5: str
    file_path: Optional[str] = None
    installed: Optional[bool] = False
    found_emulator: Optional[bool] = False
    started: Optional[bool] = False
    finished: Optional[bool] = False
    manifest: Optional[ManifestInfo | None] = None
    hooks: List = field(default_factory=lambda: [])
    ...
```

Listing 4.12: Initializing the connection to the mongo database in the constructor of the AnalyzerMongo class.

```
class AnalyzerMongo:
    def __init__(self):
        self.client = MongoClient(DBConfig.uri, server_api=ServerApi('1'))
        self.db = self.client.get_database("analyzer")
        self.coll = self.db.get_collection("samples")
        ...
```

4.1.8 Analysis

After having described the individual modules of the application, we can look at the **ApkAnalyzer** class that automates the analysis of APK files. The class initializes by starting emulators and setting up Frida instrumentation for dynamic analysis. The analyze method takes a file path and MD5 hash of an APK, extracts and stores its manifest information, and deploys it on an available emulator. It then installs the APK, applies predefined Frida hooks, simulates user interactions, and collects runtime data before uninstalling the APK. The results of the analysis, including hooks and permission details, are stored in a MongoDB database. The **get_status** method asynchronously retrieves the analysis status for a given MD5 hash, including detailed information on hooks and permissions found during the analysis. The implementation ensures thorough analysis by leveraging emulators and dynamic instrumentation while maintaining organized logging and data management. This process is initiated directly from the FastAPI handler. The prologue of the method can be seen at listing 4.13

Listing 4.13: First lines of **analyze** method from ApkAnalyzer class, illustrating the logic.

```
def analyze(self, filepath: str, md5: str) -> AnalysisResult:
    self.logger.info(f"starting to analyze sample: {filepath}")

    apk = APK(filepath.encode('utf-8'))
    self.analyzer_db.set_manifest_info(md5, ManifestInfo(
        pkn=apk.package,
        label=apk.get_app_name(),
        permissions=apk.get_permissions()
    ))

    emulator = self.emulation_pool.get_available_emulator(apk)
    if not emulator:
        self.logger.info("no open emulator found")
        return
    emulator.lock()
    self.logger.info("found emulator: {}".format(emulator.avd))
    emulator.install_sample(filepath)
    ...
    ...
    ...
```

In summary FastAPI serves as the web framework, enabling efficient communication between the server and client applications. Emulators are managed effectively through the EmulationPool, facilitating controlled analysis tasks. Frida integration allows for dynamic instrumentation, essential for extracting insights from application behavior. MongoDB handles data storage reliably, ensuring all hook event information is securely managed. Together, these components create a robust platform for comprehensive security assessment and testing in Android environments.

4.2 Front End

The proof of concept application I've developed serves as the frontend of the project, providing a live demonstration of the analyzer's capabilities in action within an Android environment. While not intended as a final product, this application is instrumental in showcasing how the analyzer functions in practical scenarios. It offers a hands-on example of how users can interact with and leverage the analyzer's features. The following sections will delve into the structure and organization of this application, highlighting its design and functionality to illustrate its role in demonstrating the analyzer's utility effectively.

The application employs two methods to detect potentially malicious applications on the user's device. The first method involves conducting a comprehensive scan of the local file system, which includes installed applications. The second method utilizes a **FileObserver**, a component that continuously monitors all filesystem changes. This observer detects new APK files appearing on the device, whether they are pushed to the SD card or downloaded from the internet, enabling proactive detection of potential security threats in real-time.

4.2.1 Permissions

The scanning tasks mentioned earlier primarily require storage-related permissions. These permissions are requested only once when the application is initially launched, ensuring seamless operation of the scanning functionalities without recurring prompts to the user. The permissions needed are listed in the **Manifest** file and are visible at figure 4.14.

Listing 4.14: Permissions listed in the manifest file of the application.

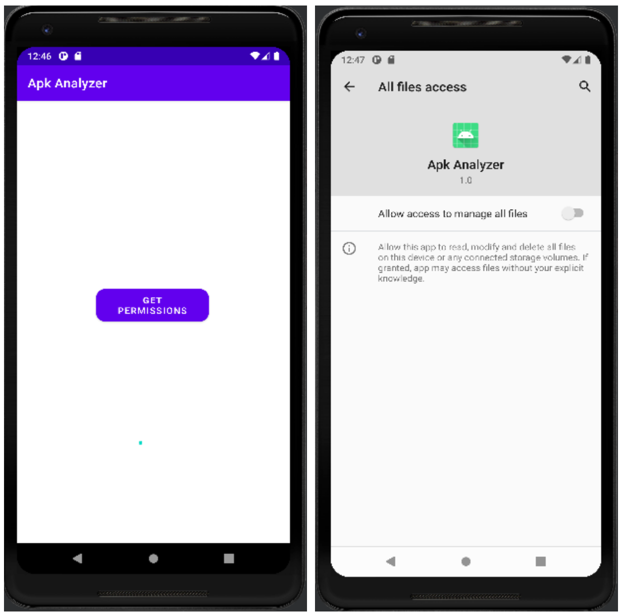
```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.MANAGE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.POST_NOTIFICATIONS"/>
```

The app asks for these permissions during runtime and the code responsible for requesting these permissions is shown in the code snippet at 4.15

Listing 4.15: Starting a new intent to ask for special permission and after asking for default read_storage and post_notifications permission.

```java
Intent intent = new Intent(Settings.ACTION_MANAGE_APP_ALL_FILES_ACCESS_PERMISSION);
Uri uri = Uri.fromParts("package", getPackageName(), null);
intent.setData(uri);
startActivity(intent);
ActivityCompat.requestPermissions(MainActivity.this,
        new String[]{Manifest.permission.READ_EXTERNAL_STORAGE,
            Manifest.permission.POST_NOTIFICATIONS},
        1);
```

This functionality translates to the following user-interface views show in figure 4.2



Allow this app to read, modify and delete all files on this device or any connected storage volumes. If granted, app may access files without your explicit knowledge.

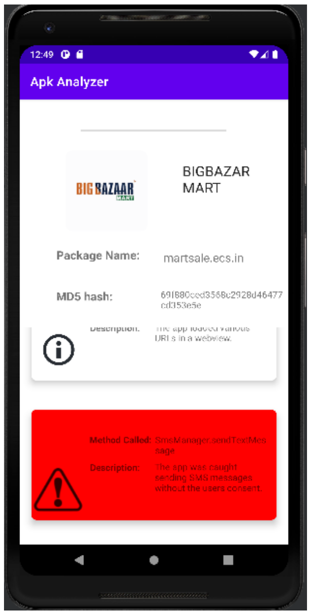
Figure 4.2: Images showing the user-interface on application start-up with button to request for permissions, manage storage access permission screen.

4.2.2 Analysis

To model the concept of an analysis I created an **Analysis** class, which is basically a thread. Upon initiation, this class transfers the selected APK file to the backend for analysis. Throughout the analysis duration, the class continually pings the server to fetch real-time progress updates. Information retrieved during these pings is stored locally, enabling the user interface to dynamically reflect the current analysis status. This approach ensures that users can monitor the analysis process closely and stay informed about its ongoing progress. The code snippet at **??** exemplifies how an Analysis is started. The user can also view the ongoing analysis 4.3.

Listing 4.16: Creating an Analysis with necessary arguments and its initiation

```java
Analysis analysis = new Analysis(file, filehash, myHandler);
analysis.start();
```



BIGBAZAR MART

Package Name: martsale.ecs.in

MD5 hash: 691880ccd3568c2928d46477cd353e5e

Description: The app loaded various URLs in a webview.

Method Called: SmsManager.sendTextMessage

Description: The app was caught sending SMS messages without the users consent.

Figure 4.3: Image showcasing the Analysis activity which essentially models the corresponding thread.

4.2.3 Filesystem scan

The main way to examine applications installed on the device is to perform a scan on the filesystem, and send the APK files that were found to the analyzzer for further dissection. The method responsible for searching for these files is shown on figure 4.17. The result of one such scan can be seen at figure 4.4 courtesy of **HomeActivity**, before and after scan completion.

Listing 4.17: Function responsible for scanning the file system and installed applications, returning a list of files upon completion.

```java
public List<File> scanForAPKs(String folderPath) {
    List<File> apkFiles = new ArrayList<>();
    File folder = new File(folderPath);
    scanFolderForAPKs(folder, apkFiles);
    scanInstalledAPKs(apkFiles);
    return apkFiles;
}
```

4.2.4 File Observer

The second, more dynamic approach to identifying potentially harmful files on the user's phone is the File Observer mechanism. Android provides developers with the **FileObserver** class, which can monitor changes to specific files or folders [13]. In my implementation, I set up a FileObserver to monitor the entire storage for changes. Specifically, I focused on the **CLOSE_WRITE** event, which triggers when the OS finishes writing to a file. Whenever this event occurs, the FileObserver sends the newly written files to the backend for examination. This can be seen in

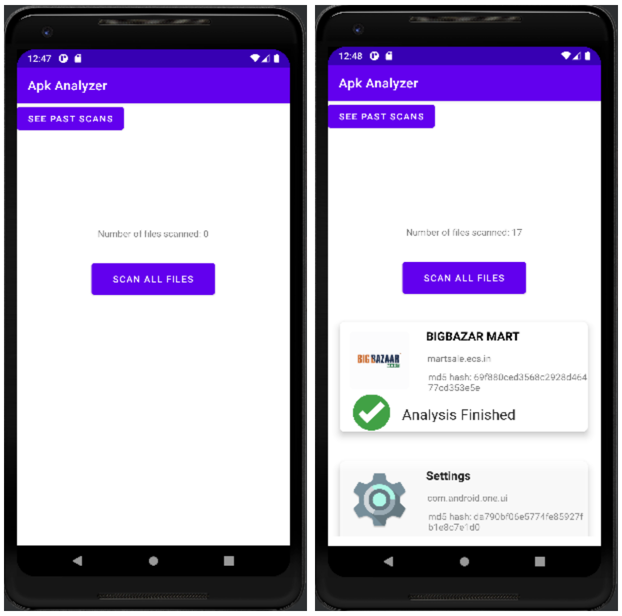


Figure 4.4: Before and after scan completion. After scan completion, found APKs are visible in the RecyclerView

listing 4.18 which shows the callback function of the Observer. Files are sent to emulation when the corresponding event is fired.

Listing 4.18: callback function of the FileObserver class, sending newly written files to analysis.

```java
public void onEvent(int event, String path) {
    switch (event) {
        case FileObserver.CLOSE_WRITE:
            Log.d(TAG, "File or directory " + path + " was closed for writing");
            String finalPath = FileObserver.parentFile + "/" + path;
            File file = new File(finalPath);
            String filehash = calculateMD5(file);
            Analysis analysis = new Analysis(file, filehash, myHandler);
            analysis.start();
        default:
            break;
    }
}

}
```

This scan-on-download feature is essential for any competent antivirus software, as it enables the real-time identification of harmful applications. By continuously monitoring for newly downloaded files and immediately scanning them upon completion, the antivirus software can promptly detect and address potential threats. This proactive approach ensures that malicious applications are identified and mitigated before they can cause any harm, significantly enhancing the security and reliability of the user's device.

4.2.5 Logging

To effectively manage and store the details and timestamps related to scans, I opted for a local Room database. Room serves as an abstraction layer over SQLite, offering a streamlined and fluent approach to database access while retaining the full capabilities of SQL [13]. This choice ensures efficient data handling and retrieval for my application. Specifically, I decided to store essential information for each scan, including **scan time, package name, and MD5 hash**. These attributes capture the critical aspects of each scan, providing a comprehensive record of scanning activities. The data entity that models this entry, designed to align with Room's database schema requirements, can be seen in detail at 4.19. This setup not only organizes the scan data systematically but also facilitates easy querying and manipulation within the application.

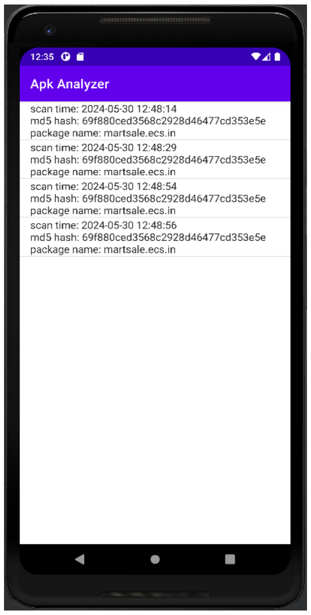
Listing 4.19: Data entity that modells an entry in the Room database, and an auxiliary toString method.

```
@Entity
public class Scan {
    @PrimaryKey(autoGenerate = true)
    public int id;

    @ColumnInfo(name = "st")
    public long scan_time;

    @ColumnInfo(name = "md5")
    public String md5;

    @ColumnInfo(name = "pkn")
    public String pkn;

    public String toString(){
        java.util.Date time = new java.util.Date((long)scan_time*1000);
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss",
            Locale.getDefault());

        return "scan time: " + sdf.format(time) + "\n" + "md5 hash: " + md5 + "\n" +
            "package name: " + pkn;
    }

}
```

To access the logs section, the user needs to click the **See past scans** button visible in figure 4.4. The information displayed to the user and it's format can be seen in figure 4.5

4.3 Running the application

The GitHub repository contains the implementations for both the front-end and back-end of the application. To build the front-end application, it is recommended to use Android Studio. I have



```
scan time: 2024-05-30 12:48:14
md5 hash: 69f880ced3568c2928d46477cd353e5e
package name: martsale.ecs.in
scan time: 2024-05-30 12:48:29
md5 hash: 69f880ced3568c2928d46477cd353e5e
package name: martsale.ecs.in
scan time: 2024-05-30 12:48:54
md5 hash: 69f880ced3568c2928d46477cd353e5e
package name: martsale.ecs.in
scan time: 2024-05-30 12:48:56
md5 hash: 69f880ced3568c2928d46477cd353e5e
package name: martsale.ecs.in
```

Figure 4.5: Activity associated with displaying information related to past scans. The data is split up into a ListView.

successfully built it using this method. Once built, the resulting APK file can be installed on an Android device. You can transfer the APK to your device's storage and install it via the file manager or use ADB. Upon first launch, ensure you accept the requested permissions for the application to function properly. The FileObserver should then be active, and you can perform a manual scan using the Scan button if desired.

To run the back-end, start by installing the dependencies listed in the requirements file using pip. Next, configure the application by modifying the files in the config package. Essential configurations include setting the path to the ADB executable, the MongoDB connection string, and the path to the emulator executable along with the used images. Other configurations are available, but these are mandatory. After configuring the analyzer, you can start the application by running the main.py file with Python. The setup should complete in approximately 1-2 minutes, after which the analyzer will be ready for use.

5 Testing and experimental results

The validation of results is a crucial aspect of any research project, ensuring the accuracy, reliability, and credibility of the findings. In our study, the validation was primarily manual, involving running the program on various APK files to verify functionality and making necessary modifications when issues arose. Manual validation provides a human perspective essential for assessing usability and overall user experience, and allows for a more exploratory approach compared to automated testing, which follows predetermined scripts and may miss unexpected issues. Some of the samples used were provided by Bitdefender, which contributed to the robustness of our validation approach [24].

The program was designed to analyze APK files, extracting detailed information about their contents, including decompiling APKs, analyzing their structure, and identifying potential security issues. Challenges included handling the variety of APK file structures and integrating third-party libraries, which introduced compatibility issues. Manual testing helped identify and rectify these issues promptly, ensuring the tool's accuracy and reliability. Despite the challenges, manual validation played a critical role in ensuring the comprehensive testing of the application, providing the flexibility to adapt and address unforeseen issues, and ultimately ensuring that the final product is of high quality and user-friendly .

5.1 Encountered dificulties

During the development and validation process, several challenges were encountered that significantly impacted the workflow and required meticulous troubleshooting. One major issue was improper hook loading, where incorrect methods were hooked, leading to unexpected behaviors and inaccuracies in the analysis. Additionally, certain hooks caused the applications to crash during analysis, necessitating careful adjustments and retries. Communication between the Android Debug Bridge (ADB) and the emulator frequently failed, disrupting the analysis process and requiring repeated connections. These difficulties underscored the complexity of the validation process and the importance of robust error handling mechanisms.

5.1.1 Hooks

As outlined in [20], Frida is definitely not flawless. The paper found that Frida is often very resource-consuming, an aspect that is aggravated in the case of emulators. The fact that it is resource-intensive, combined with the frequent developer complaints about errors with certain hooks, underscores the potential issues. These errors include improper hook loading, where incorrect methods are hooked, leading to unreliable analysis results and system instability. Moreover, some hooks caused applications to crash during the analysis, further complicating the testing process. These challenges necessitate robust error handling and careful debugging to ensure accurate and reliable outcomes. Understanding and mitigating these issues is crucial for the effective use of Frida in application analysis.

One particular hook that often caused application crashes was the **load** and **loadLibrary** methods of the System class. These methods are responsible for loading native libraries, as detailed in [25]. Native libraries play a significant role in Android malware, as they are often used to execute malicious code. Understanding which native libraries are loaded during runtime can greatly aid in the analysis of potentially malicious applications. By tracking these libraries, analysts can gain insights into the behavior and potential threats posed by the application, thus improving the overall effectiveness of the malware detection process.

The code listing found at 5.1 was my initial implementation of the hook.

Listing 5.1: Frida hook written in JavaScript to hook loadLibrary method and forward information from it to frida server.

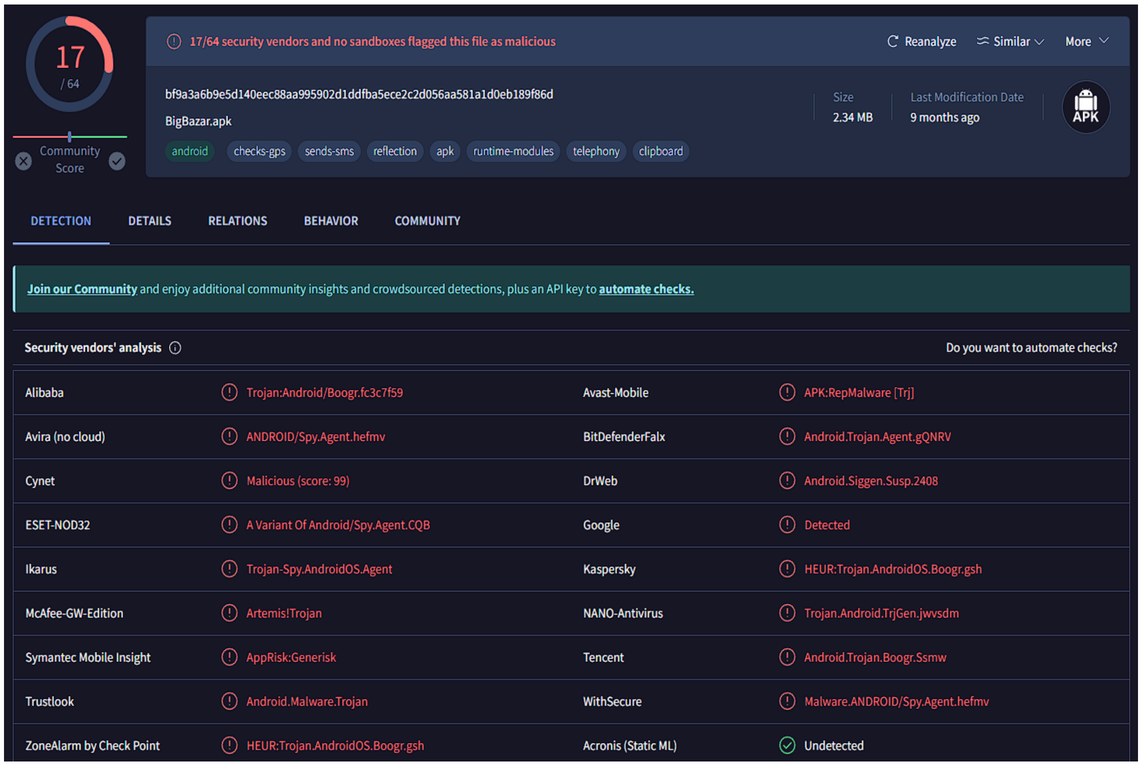
```
var System = Java.use('java.lang.System');
var Runtime = Java.use('java.lang.Runtime');
var VMStack = Java.use('dalvik.system.VMStack');

System.loadLibrary.implementation = function(library) {
    var timestamp = new Date().getTime();
    var hookEventData = {
        type: 'library',
        method: 'System.loadLibrary',
        timestamp: timestamp,
        args: [library],
        return_value: null
    };
    send(JSON.stringify(hookEventData));
    return this.loadLibrary(library);

};
```

After thorough research, I discovered that the load and loadLibrary methods should not be hooked using conventional hooking frameworks because they rely on the class loader of the application. This dependency on the class loader introduces complications and instability when attempting to hook these methods. Despite various attempts to resolve this issue, I was unable to achieve a stable solution. Consequently, the most viable workaround was to remove the hook entirely. This approach, while not ideal, ensured that the application remained functional and stable, allowing the analysis to proceed without unnecessary interruptions.

One such malware that loads native libraries is the APK file associated with the MD5 hash **69f880ced3568c2928d46477cd353e5e**. This sample is identified as malicious by 17 security vendors, most of which classify it as a **Trojan** as shown in this VirusTotal report 5.1. If I had left the library hook enabled during the analysis of this APK, the process would have halted as soon as the first native library was loaded, rendering the entire analysis ineffective. However, by disabling the hook, I was able to proceed with the analysis and obtain relevant hook results, ensuring a comprehensive examination of the malware's behavior.

5.1.2 Emulator and ADB

During the development and analysis process, significant challenges were encountered with the Android Debug Bridge (ADB) and emulators. A common issue was the frequent loss of connection between ADB and the emulator, which required constant reconnections and hindered workflow efficiency. Additionally, certain ADB commands would occasionally cause the emulator to crash, further complicating the testing process.



17/64 security vendors and no sandboxes flagged this file as malicious

bf9a3a6b9e5d140eec88aa995902d1ddfba5ece2c2d056aa581a1d0eb189f86d
BigBazar.apk

android checks-gps sends-sms reflection apk runtime-modules telephony clipboard

Size 2.34 MB
Last Modification Date 9 months ago

Join our Community and enjoy additional community insights and crowdsourced detections, plus an API key to automate checks.

Security vendors' analysis

Do you want to automate checks?

Alibaba	Trojan:Android/Boogr.fc3c7f59	Avast-Mobile	APK:RepMalware [Trj]
Avira (no cloud)	ANDROID/Spy.Agent.hefmv	BitDefenderFalx	Android.Trojan.Agent.gQNRV
Cynet	Malicious (score: 99)	DrWeb	Android.Siggen.Susp.2408
ESET-NOD32	A Variant Of Android/Spy.Agent.CQB	Google	Detected
Ikarus	Trojan-Spy.AndroidOS.Agent	Kaspersky	HEUR:Trojan.AndroidOS.Boogr.gsh
McAfee-GW-Edition	Artemis!Trojan	NANO-Antivirus	Trojan.Android.TrjGen.jwvsdm
Symantec Mobile Insight	AppRisk:Generisk	Tencent	Android.Trojan.Boogr.Ssmw
Trustlook	Android.Malware.Trojan	WithSecure	Malware.ANDROID/Spy.Agent.hefmv
ZoneAlarm by Check Point	HEUR:Trojan.AndroidOS.Boogr.gsh	Acronis (Static ML)	✓ Undetected

Figure 5.1: VirusTotal overview of sample with 17 security vendors detecting it as malicious .

To adress these issues, I have implemented a few mechanisms to prevent these crahes. One such mechanism was waiting for the emulator to completely start at startup. The method responsible for determining whether the emulator booted is found at 5.2.

Listing 5.2: Method responsible for determining if emulator is actually running or not, with the help of ADB.

```
@property
def is_running(self):
    output, _ = self.adb.shell(["getprop", "init.svc.bootanim"])
    is_running = True if output.decode('utf-8').strip() == 'stopped' else False
    return is_running
```

Another mechanism to address this issue was to monitor and terminate the Jester process associated with an emulator. The Jester process heavily relies on ADB, so reducing its runtime can improve reliability. By ensuring that the Jester process is killed whenever an analysis finishes, we prevent it from running unnecessarily and causing potential disruptions. By trying to kill the Jester whenever an analysis finishes ensures it is not running pointlessly 5.3.

Listing 5.3: Method responsible for waiting Jester process to conclude.

```
def wait_for_clown(self):
    self.clown.join()
```

These mechanisms do not completely eliminate the presence of such bugs and errors, but they greatly reduce their frequency of occurrence. By implementing these strategies, the stability and reliability of the analysis process are significantly improved.

5.2 Results

To determine the effectiveness of the analyzer, it's important to examine the actual results. This can be approached in multiple ways. One method is to analyze the heuristics generated by running the analyzer on various samples. For this purpose, I decided to evaluate the MongoDB collection used during development. Another approach is to look at specific samples and examine the results produced by the analyzer. This dual method allows for a comprehensive assessment of the analyzer's performance and accuracy.

5.2.1 MongoDB statistics

One metric of great interest is identifying the most common hooks. From a developer's standpoint, this helps determine which areas of the application need enhancement or what types of hooks would be beneficial. Figure 5.2 illustrates the five most popular hooks, showing that the init method of the DexFile class has over 350 occurrences. This frequency has the potential to generate significant spam. To address this, we could introduce a mechanism to limit the number of redundant hooks recorded for a given sample, avoiding multiple entries for the same dex file.

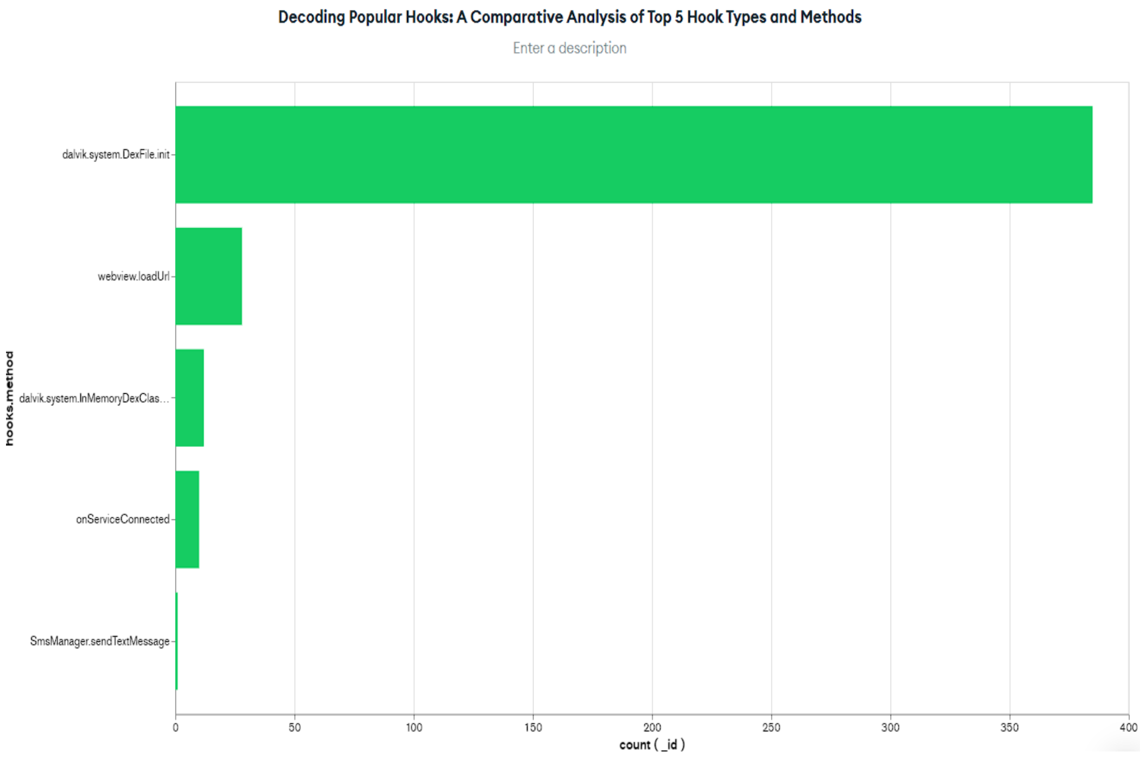


Figure 5.2: 5 most common hooks caught by analyzer. x axis shows the count, y axis represents the unique hooks

Another interesting statistic to examine is the success rate of file installations on the emulator. This metric helps identify potential compatibility issues with certain API versions or architectures. Figure 5.3 shows that only three samples were not installed successfully. While this is not a significant issue at the moment, it highlights an area for potential improvement. To further reduce the number of failed installations, it would be beneficial to add more emulators with different SDK levels and architectures. This expansion would enhance support for a broader range of APKs and ensure comprehensive compatibility testing.

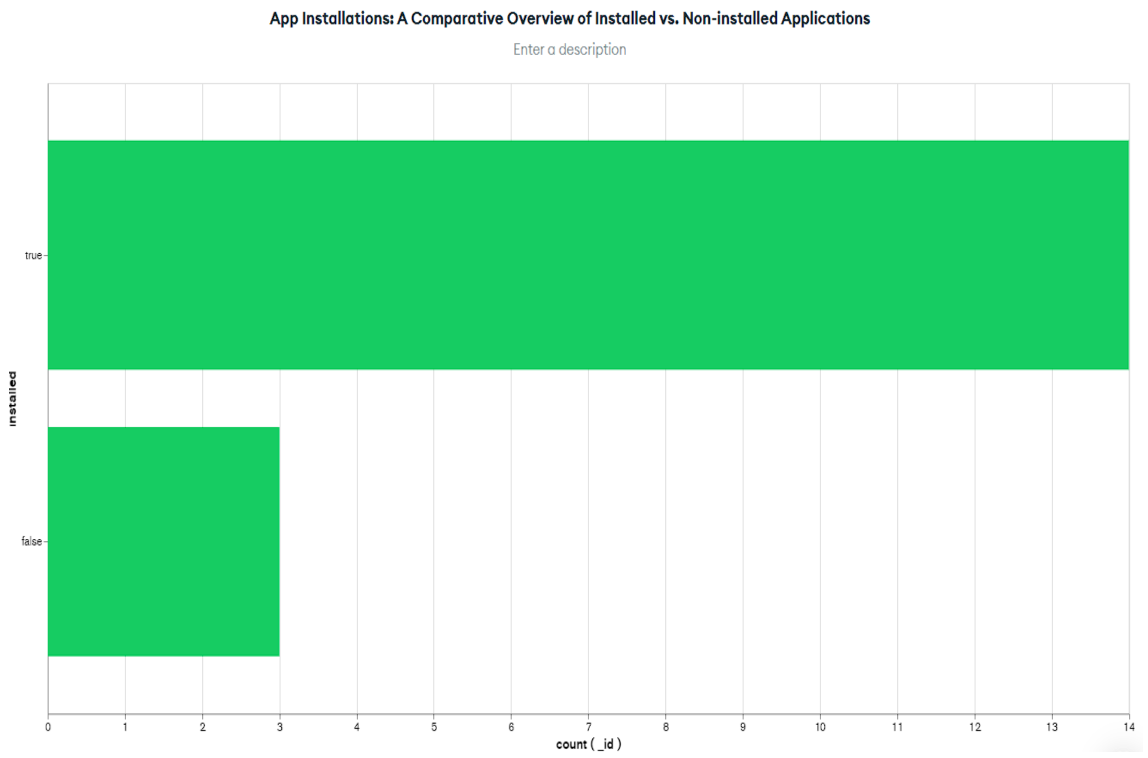


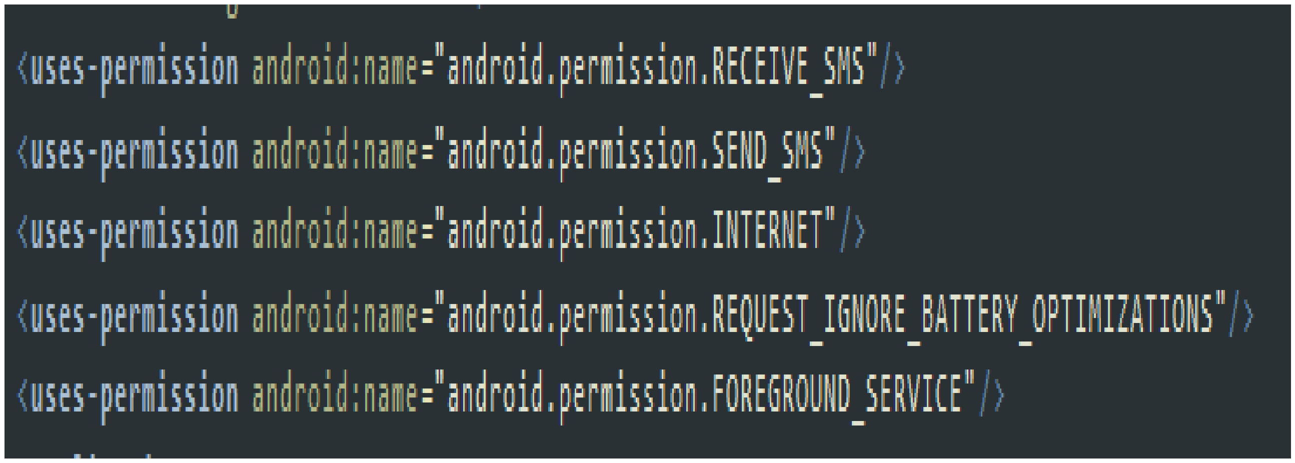
Figure 5.3: Figure that shows how many apps were successfully installed by the analyzer. X axis shows the count and the Y axis shows if it was installed or not; true means it was installed and false means it was not installed.

5.2.2 Case study

To thoroughly evaluate the quality of a sample's analysis, it is crucial to conduct a comprehensive analysis on a particular sample. This involves examining the specific hooks that were captured during the analysis and comparing them with the hooks that should have been detected. By scrutinizing the differences, we can assess the accuracy and effectiveness of the analyzer. This section aims to provide an in-depth evaluation of these aspects by focusing on a specific sample, highlighting both the strengths and areas for improvement in the analysis process. Through this detailed investigation, we can gain valuable insights into the performance and reliability of the analyzer, ultimately enhancing its capability to accurately identify and capture relevant hooks.

The sample we'll focus on is the one shown at 5.1, an APK file with the MD5 **69f880ced3568c2928d46477cd353e5e**.

Before we process the sample with the analyzer, I will perform a comprehensive manual static analysis to elucidate the malware's functionality. This preliminary step is crucial for understanding the inherent behaviors and permissions requested by the application, which may indicate malicious intent. For this analysis, I will employ the **Jadx** decompiler, a powerful tool that converts the APK's compiled code into a human-readable format, allowing for detailed inspection of the application's structure and code. Figure 5.4 illustrates the permissions declared in the sample's Android manifest file. By examining these permissions, we can discern the types of operations the application is authorized to perform. Notably, the manifest includes permissions related to sending text messages, which is a significant red flag. Such permissions can be exploited by malware to send unauthorized SMS messages, potentially leading to unauthorized charges or the dissemination of spam and phishing messages.



```
<uses-permission android:name="android.permission.RECEIVE_SMS"/>
<uses-permission android:name="android.permission.SEND_SMS"/>
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.REQUEST_IGNORE_BATTERY_OPTIMIZATIONS"/>
<uses-permission android:name="android.permission.FOREGROUND_SERVICE"/>
```

Figure 5.4: Figure showing permissions listed in malware samples Android Manifest file.

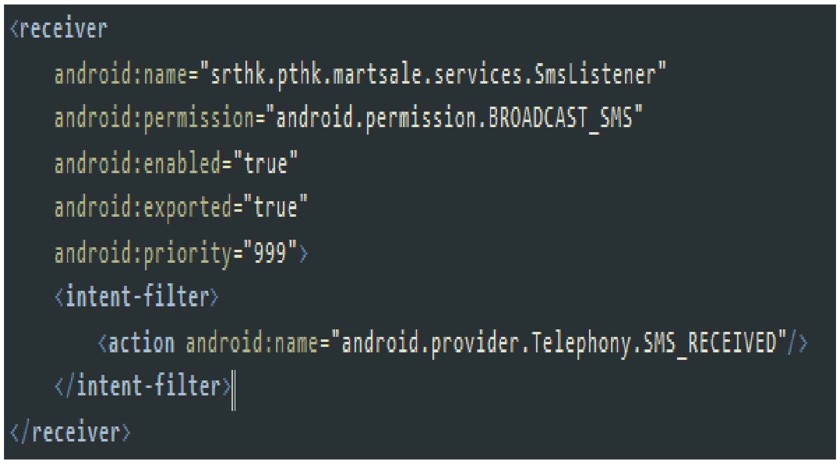
Upon further examination of the manifest file, we observe the declaration of an SMS receiver, as shown in Figure 5.5. The implications of such functionality are significant. With an SMS receiver in place, the application can intercept and process all incoming text messages. This capability allows the application to access the contents of the messages, including any sensitive information they may contain, such as authentication codes, personal messages, or contact details. Additionally, the application can also determine the sender's information, which can be used for various malicious purposes, such as spoofing or targeted phishing attacks.

Next, we will closely examine the SMS receiver by double-clicking on its name within the decompiled code. This action will allow us to inspect the specific implementation details of the receiver. Upon thorough inspection, it becomes evident that the program is configured to forward any received SMS messages to a custom number, which is stored in the shared preferences file, as illustrated in 5.6.

This finding is particularly alarming as it confirms the malicious intent of the application. By forwarding incoming SMS messages to a predefined number, the application effectively exfiltrates potentially sensitive information without the user's consent.

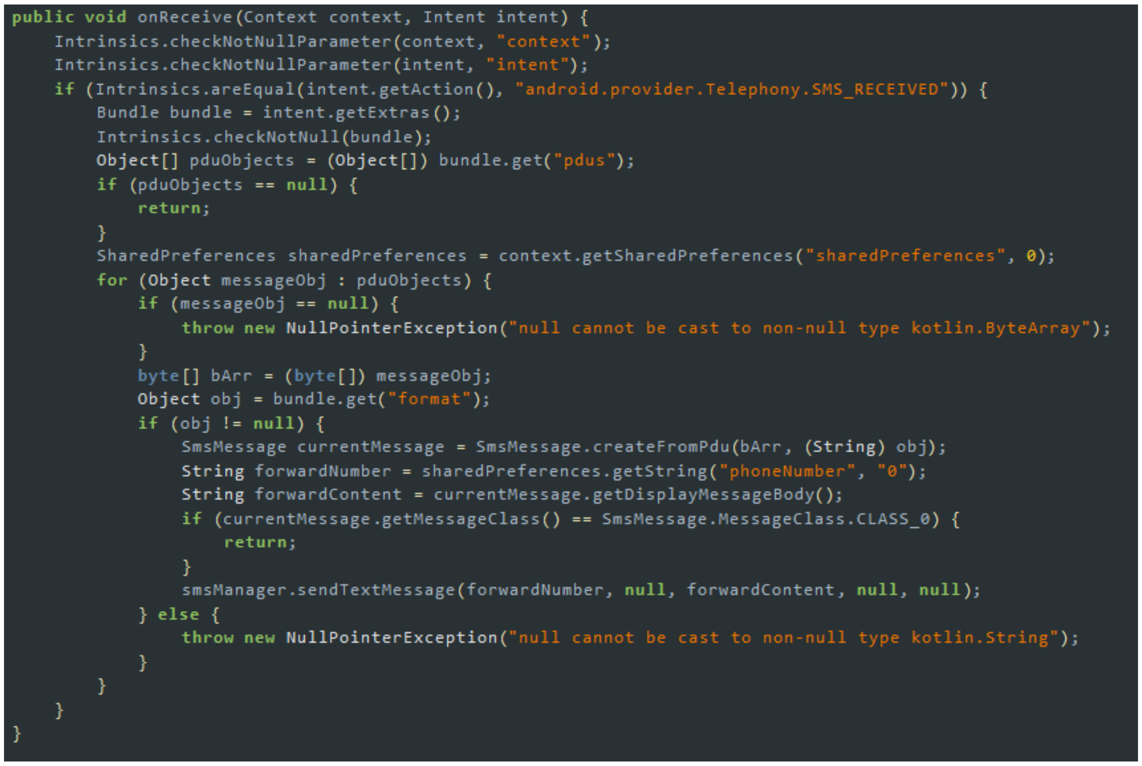
Next, we shall proceed to process this sample using our analyzer. The first step involves installing the front-end application on a device, ensuring that all necessary permissions are granted for it to function effectively. With the permissions in place, our analyzer is primed and ready to operate.

To initiate the analysis, we use the **adb push** command to transfer our sample APK file to the device's storage. The front-end application is designed to automatically detect the presence



```
<receiver
    android:name="srthk.pthk.martsale.services.SmsListener"
    android:permission="android.permission.BROADCAST_SMS"
    android:enabled="true"
    android:exported="true"
    android:priority="999">
    <intent-filter>
        <action android:name="android.provider.Telephony.SMS_RECEIVED"/>
    </intent-filter>
</receiver>
```

Figure 5.5: Figure showing SMS receiver being declared in manifest file of sample.



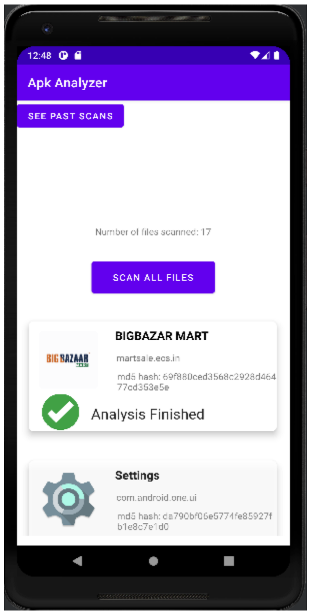
```
public void onReceive(Context context, Intent intent) {
    Intrinsics.checkNotNullParameter(context, "context");
    Intrinsics.checkNotNullParameter(intent, "intent");
    if (Intrinsics.areEqual(intent.getAction(), "android.provider.Telephony.SMS_RECEIVED")) {
        Bundle bundle = intent.getExtras();
        Intrinsics.checkNotNull(bundle);
        Object[] pduObjects = (Object[]) bundle.get("pdus");
        if (pduObjects == null) {
            return;
        }
        SharedPreferences sharedPreferences = context.getSharedPreferences("sharedPreferences", 0);
        for (Object messageObj : pduObjects) {
            if (messageObj == null) {
                throw new NullPointerException("null cannot be cast to non-null type kotlin.ByteArray");
            }
            byte[] bArr = (byte[]) messageObj;
            Object obj = bundle.get("format");
            if (obj != null) {
                SmsMessage currentMessage = SmsMessage.createFromPdu(bArr, (String) obj);
                String forwardNumber = sharedPreferences.getString("phoneNumber", "0");
                String forwardContent = currentMessage.getDisplayMessageBody();
                if (currentMessage.getMessageClass() == SmsMessage.MessageClass.CLASS_0) {
                    return;
                }
                smsManager.sendTextMessage(forwardNumber, null, forwardContent, null, null);
            } else {
                throw new NullPointerException("null cannot be cast to non-null type kotlin.String");
            }
        }
    }
}
```

Figure 5.6: Receiver of sample, showing how the received SMS is forwarded to a custom number.

of new files in the specified directory. Once the sample is detected, the analysis process begins immediately. After analysis is finished we are greeted with figure 5.7

Among the various hooks captured, the most significant finding is the program's action of forwarding text messages. This malicious behavior is prominently displayed in red to draw immediate attention. Figure 5.8 illustrates this finding clearly, showing that the analyzer successfully intercepted and logged the suspicious activity.

This detailed view not only confirms the presence of malicious functionality but also pro-



12:48

Apk Analyzer

SEE PAST SCANS

Number of files scanned: 17

SCAN ALL FILES

BIGBAZAR MART

martsale.ecs.in

md5 hash: 69f880ced3568c2928d464
77cd353e5e

Analysis Finished

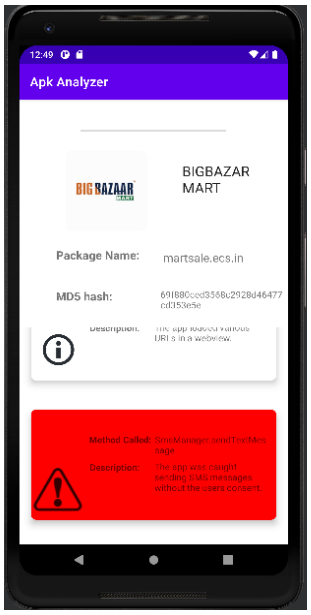
Settings

com.android.one.ui

md5 hash: da790bf06e5774fe85927f
b1e8c7e1d0

Figure 5.7: Scanning finished on device, with sample being shown on the main page with a check mark.

vides a clear and straightforward understanding of how the application operates and what specific actions it performs. By offering such targeted insights, the analyzer equips users with the necessary information to make informed decisions about the security of the APK file and take appropriate measures to protect their devices.



12:49

Apk Analyzer

BIGBAZAR
MART

Package Name: martsale.ecs.in

MD5 hash: 69f880ced3568c2928d46477
cd353e5e

Description: The app loaded various URLs in a webview.

Method Called: SmsManager.sendTextMessage

Description: The app was caught sending SMS messages without the users consent.

Figure 5.8: Image showcasing detailed results of analysis.

This case study demonstrates the effectiveness of our APK analyzer in identifying and analyzing malicious behaviors within Android applications. Through comprehensive static and dynamic analysis, we identified significant security risks, including the program's behavior of forwarding received SMS messages to a custom number.

The dynamic analysis was enhanced by the Jester module, which successfully accepted the SEND_SMS permission, allowing the malware to execute its intended method. This enabled the analyzer to capture critical hooks and provide clear descriptions, confirming the presence of malicious actions.

The detailed analysis screen offered a clear understanding of the malware's functionality, underscoring the importance of dynamic analysis in uncovering sophisticated threats. This case study validates that the application works correctly, effectively detecting and documenting malware.

The study also demonstrated that each module performed its specific tasks as intended, showcasing the system's robustness and reliability.

5.3 Results compared to other products

To validate my results it's essential to compare my application to other application in order to assess its effectiveness. Since I did not manage to get AndroPyTool working, I will be comparing my results to **MobSF**. The results of the analysis done by MobSF can be found at `https://mobsf.live/static_analyzer/69f880ced3568c2928d46477cd353e5e/`.

Unfortunately I could not get the dynamic analyzer of MobSF working so again, I will be strictly referring to static analysis results.

The static analysis features of MobSF are more complex than those in my application, offering a broader range of detailed insights into the APK. MobSF provides an extensive breakdown of permissions, potential security issues, code analysis, and more.

The implementation of ApkAnalyzer offers a significant advantage by incorporating a user-side Android application, allowing real-time monitoring and interaction during the analysis process. This feature sets ApkAnalyzer apart from many traditional static and dynamic analysis tools, which often operate in a more isolated and less interactive manner. The user-side application provides immediate feedback to users by displaying the results of the analysis as they are generated. This real-time capability is particularly beneficial for users who need to quickly assess the safety and integrity of an APK before installation or during its operation. By offering live insights, ApkAnalyzer enhances the user's ability to make informed decisions based on the latest analysis data.

Additionally, the user-side application facilitates a more user-friendly and accessible experience, enabling non-technical users to benefit from advanced security analysis without requiring deep technical knowledge. Users can view detailed reports, track the progress of the analysis. Another key benefit is the integration of a file observer, which continuously monitors the file system for any new or modified APK files. This ensures that any changes are immediately analyzed, providing proactive protection against potential threats. While MobSF offers a comprehensive analysis with a detailed UI, ApkAnalyzer's real-time updates and immediate file observation capabilities enhance its practicality and usability in real-world scenarios. By bridging the gap between complex analysis processes and user accessibility, ApkAnalyzer ensures that comprehensive security assessments are available to a broader audience, thereby promoting better security practices and awareness.

5.4 Performance

The performance of the ApkAnalyzer application is a critical aspect of its overall effectiveness and user satisfaction. In this subsection, we detail the methodologies and results of our performance validation efforts, focusing on various key metrics such as analysis time, resource usage, and scalability.

5.4.1 Analysis Time

The duration of the analysis process is predominantly influenced by two critical factors. The first and more variable factor is the parsing of the APK file using Androguard. The parsing phase is heavily contingent on the size of the APK file. Larger APK files contain more data and require more time to parse, while smaller files can be processed more quickly. This dependency on file size makes the parsing time an important and sometimes unpredictable aspect of the overall analysis duration.

The second factor influencing the analysis time is the actual analysis process itself, which is generally more consistent. The time taken for this phase is largely governed by a configurable parameter within the analysis system. This parameter sets a predetermined duration for the analysis, ensuring that it adheres to specific time constraints. Users or administrators can modify this configuration value based on their needs, allowing for flexibility in how long the analysis takes. While this component of the analysis time is essentially fixed, its adjustability means that it can be tailored to accommodate various operational requirements or constraints.

To accurately measure the duration of the parsing process, I utilized Python's built-in **time** package. This package provides various functions to handle time-related tasks, making it an ideal choice for this purpose. In particular, I employed it to capture the start and end times of the parsing operation, allowing for precise calculation of the elapsed time. Listing 5.4 illustrates the code used to time the parsing process, showcasing the method I implemented to ensure accurate timing.

Listing 5.4: Code snippet that times the parsing time of APK file by Androguard.

```
apk_parsing = time.time()
apk = APK(filepath.encode('utf-8'))
self.logger.info("apk parsing for md5 {} took {}".format(md5, time.time() -
    apk_parsing))
```

To assess the impact of APK size on parsing time, I selected APK files of varying sizes. The primary criterion for selection was the size of the APK, as this is a significant factor influencing the parsing duration. The chosen samples are identified by their respective MD5 checksums: **6a2d5c0f05c9307c3c61a387c4c4d54c, 69f880ced3568c2928d46477cd353e5e, 5d6ac6d128b29b2e6bbdb289568d02f6**. Among these, the largest APK file is **5d6ac6d128b29b2e6bbdb289568d02f6**, which corresponds to the Coinbase application and has a size of 66MB. The other two samples are significantly smaller, with sizes of 3MB and 45MB, respectively.

The parsing times for each of these samples are documented in Listing 5.5

Listing 5.5: Time of parsing APK measured for different MD5s. Timings taken from logging.

```
INFO: 2024-06-13 12:09:52,430 - apk_analyer - apk parsing for md5
    6a2d5c0f05c9307c3c61a387c4c4d54c took 0.4101226329803467
...
INFO: 2024-06-13 12:09:51,810 - apk_analyer - apk parsing for md5
    69f880ced3568c2928d46477cd353e5e took 0.05954098701477051
...
INFO: 2024-06-13 12:24:50,667 - apk_analyer - apk parsing for md5
    5d6ac6d128b29b2e6bbdb289568d02f6 took 0.9607915878295898
```

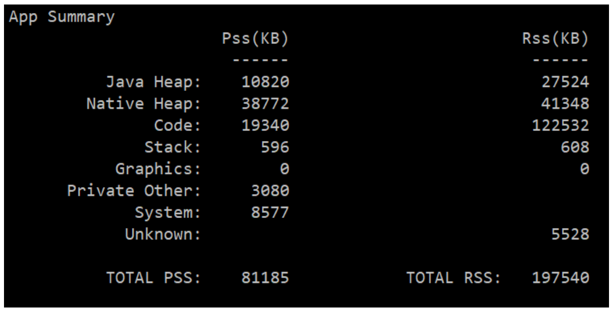
Based on these timings, we can conclude that the parsing of the APK file has a minimal impact on the overall analysis time. The longest analysis time recorded was 0.96 seconds, which remains negligible when considering the total instrumentation time of 25 seconds. This finding indicates that the parsing phase, despite being an essential step, does not significantly affect the efficiency of the analysis process. The disparity between the parsing time and the instrumentation time highlights the relative simplicity of parsing operations compared to the complexity involved in instrumenting the APK.

5.4.2 Utilized resources

A critical aspect of any competent antivirus software is its performance in terms of resource utilization. Excessive resource consumption can lead to increased costs and a degraded user experience.To evaluate the performance of my client-side application, I measured both memory and CPU usage. These metrics were measured on a Pixel 2 API 30 emulator equipped with 2GB of RAM.

To determine memory usage, I utilized the following ADB command: **adb -s emulator-5560 shell dumpsys meminfo 5271**, where the last number represents the PID of the Apk-Analyzer running on the phone. This command provides a comprehensive breakdown of the memory consumption of the application.

As illustrated in Figure 5.9 , the total memory usage of the application is approximately **81 MB**. This is within acceptable limits considering that modern smartphones typically come equipped with over 8 GB of RAM. Such a memory footprint ensures that the application runs efficiently without causing significant strain on the device's resources.



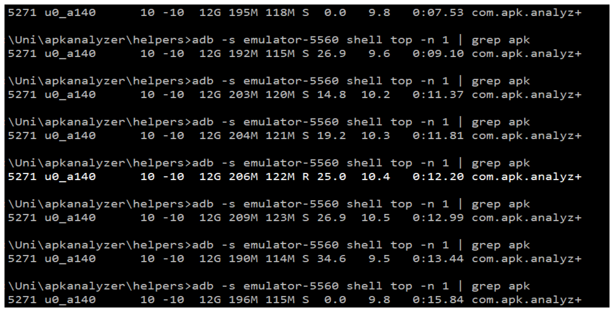
```
App Summary
                       Pss(KB)                        Rss(KB)
                        ------                         ------
           Java Heap:    10820                          27524
         Native Heap:    38772                          41348
                Code:    19340                         122532
               Stack:      596                            608
            Graphics:        0                              0
       Private Other:     3080
              System:     8577
             Unknown:                                    5528

           TOTAL PSS:    81185            TOTAL RSS:   197540
```

Figure 5.9: Figure showing information regarding memory usage of **com.apk.analyzer**.

To measure CPU usage, I utilized the **top** tool by executing the following command: **adb -s emulator-5560 shell top**. This command provides a real-time overview of CPU usage for processes running on the device.

During idle periods, the CPU usage of the application remained consistently at 0.0%, indicating minimal resource consumption when the application is not actively processing any tasks. This is crucial for ensuring that the application does not unnecessarily burden the device when it is not in use.

However, when a new analysis is initiated on a sample that has not been previously processed, the CPU usage can spike significantly. As depicted in Figure5.10 the CPU usage can rise as high as 30%. This increase in CPU utilization is expected due to the intensive nature of the analysis process, namely the continous pinging of the server side application.



```
5271 u0_a140      10 -10  12G 195M 118M S  0.0   9.8   0:07.53 com.apk.analyz+

\Uni\apkanalyzer\helpers>adb -s emulator-5560 shell top -n 1 | grep apk
5271 u0_a140      10 -10  12G 192M 115M S 26.9   9.6   0:09.10 com.apk.analyz+

\Uni\apkanalyzer\helpers>adb -s emulator-5560 shell top -n 1 | grep apk
5271 u0_a140      10 -10  12G 203M 120M S 14.8  10.2   0:11.37 com.apk.analyz+

\Uni\apkanalyzer\helpers>adb -s emulator-5560 shell top -n 1 | grep apk
5271 u0_a140      10 -10  12G 204M 121M S 19.2  10.3   0:11.81 com.apk.analyz+

\Uni\apkanalyzer\helpers>adb -s emulator-5560 shell top -n 1 | grep apk
5271 u0_a140      10 -10  12G 206M 122M R 25.0  10.4   0:12.20 com.apk.analyz+

\Uni\apkanalyzer\helpers>adb -s emulator-5560 shell top -n 1 | grep apk
5271 u0_a140      10 -10  12G 209M 123M S 26.9  10.5   0:12.99 com.apk.analyz+

\Uni\apkanalyzer\helpers>adb -s emulator-5560 shell top -n 1 | grep apk
5271 u0_a140      10 -10  12G 190M 114M S 34.6   9.5   0:13.44 com.apk.analyz+

\Uni\apkanalyzer\helpers>adb -s emulator-5560 shell top -n 1 | grep apk
5271 u0_a140      10 -10  12G 196M 115M S  0.0   9.8   0:15.84 com.apk.analyz+
```

Figure 5.10: Figure showing information regarding cpu usage of **com.apk.analyzer**. First command was fired before starting analysis, last command was given when analysis already finished.

The CPU usage could be reduced by modifying the heartbeat interval - the interval at which the user application continuously pings the server-side analyzer, or by reducing the size of the messages transferred between the 2.

5.4.3 Scalability

Scalability is a critical consideration in the design and implementation of any software application, particularly those intended for widespread deployment like the ApkAnalyzer. Ensuring that the application can handle increased loads efficiently without degradation in performance is essential for maintaining a seamless user experience and meeting the demands of a growing user base.

One of the key features that support the scalability of the ApkAnalyzer is the implementation of the Emulation Pool. This component manages a pool of Android emulators, dynamically allocating available emulators to handle incoming analysis requests. By efficiently distributing the workload across multiple emulators, the application can process multiple APK files concurrently. This horizontal scaling approach ensures that the system can handle a higher number of simultaneous analyses without bottlenecks or performance degradation.

The ApkAnalyzer is designed with a modular architecture, facilitating the straightforward addition of new hooks. This design ensures that the system can evolve without requiring extensive rework or disrupting existing functionalities. The hooks are organized within a dedicated package, frida.hooks, where each hook is implemented as a separate JavaScript file according to Frida standards.

6 Conclusions and future work

This study underscores the pivotal role of dynamic analysis in the development of a robust APK analyzer for detecting and mitigating Android malware. Leveraging advanced tools such as Frida for dynamic instrumentation and AndroidViewClient for automated user interaction simulation, the project highlights the effectiveness of real-time behavior monitoring in identifying malicious activities within Android applications.

The project's comprehensive approach to dynamic analysis not only enhances malware detection capabilities but also provides a deeper understanding of potential security threats. This real-time analysis framework enables a proactive stance in safeguarding against evolving malware strategies, making it an essential tool for developers and security researchers alike.

6.1 Results

The development and deployment of the APK analyzer have brought forth several key realizations and advantages, which collectively enhance the security and reliability of Android applications. The primary strengths of this application stem from its dynamic analysis capabilities, user interaction simulation, and comprehensive data management. These features contribute to a robust toolset for malware detection and analysis.

One of the most significant realizations is the importance of monitoring an application's behavior in real-time. Dynamic analysis provides a deeper understanding of how an application interacts with the system, revealing malicious activities that static analysis might overlook. The application's design anticipates future needs by allowing easy integration of new technologies and methodologies. This forward-thinking approach ensures that the APK analyzer remains relevant and effective as new types of malware emerge.

The incorporation of automated user interaction simulation using AndroidViewClient proved crucial in triggering and observing the full spectrum of an application's behavior. This approach ensures that even behaviors dependent on user actions are analyzed, providing a comprehensive security assessment.

Utilizing MongoDB for data management has demonstrated the application's capability to handle large volumes of analysis data efficiently. This scalability is essential for analyzing numerous applications simultaneously without compromising performance.

The application's ability to provide real-time updates during the analysis process enhances its utility. Users and researchers can monitor the analysis progress and receive immediate feedback, facilitating timely interventions and decision-making.

6.2 Future Work

While the APK analyzer has made significant strides in dynamic malware analysis, there are several areas where future enhancements can further elevate its capabilities. These potential improvements focus on expanding the scope of information captured, improving detection evasion techniques, resolving emulator-related issues, and enhancing the automated interaction system.

One of the most immediate enhancements involves extending the range of hooks used in the dynamic analysis. By incorporating more hooks, the analyzer can capture a broader array of data points, such as detailed monitoring of network communications, tracking modifications to the file system, capturing memory usage patterns, and monitoring a wider variety of API calls. These additional hooks will enable the analyzer to provide a more comprehensive assessment of an application's behavior.

Furthermore, malware often includes sophisticated techniques to detect and evade sandbox environments. To counteract this, future work should focus on implementing advanced hooking techniques that mask the presence of the analyzer, making it more difficult for malware to detect the sandbox environment. Enhancing the emulator to more accurately simulate real-world devices, including mimicking network conditions, hardware configurations, and user behavior, is also essential. Additionally, integrating hooks that modify system responses to queries commonly used by malware to detect virtual environments will further improve the analyzer's ability to evade detection.

The Jester component, which simulates user interactions, can be significantly enhanced to handle a wider variety of emulators, screens, and situations. This includes expanding support to include a broader range of emulator configurations, developing more sophisticated scripts to simulate complex user behaviors, and implementing machine learning algorithms to dynamically adapt the interaction patterns based on the application's responses. Improving the robustness of the Jester to handle unexpected situations, such as crashes or unresponsive elements, will ensure continuous and accurate analysis.

Another promising direction for future work is the integration of machine learning techniques to enhance anomaly detection. By analyzing patterns in the captured data, machine learning models can identify subtle and complex indicators of malicious behavior that may not be evident through rule-based analysis. Additionally, expanding the analyzer to support additional mobile platforms, such as iOS, would significantly broaden its applicability and utility. This involves developing platform-specific hooks and analysis techniques tailored to the unique characteristics and security models of these operating systems.

In conclusion, the APK analyzer has laid a strong foundation for dynamic malware analysis, but numerous opportunities for enhancement remain. By adding more hooks, implementing evasion techniques, resolving emulator-related issues through containerization, and improving automated interactions, the analyzer can become an even more powerful tool in the fight against mobile malware. Additionally, integrating machine learning and expanding platform support will ensure the analyzer remains at the forefront of mobile security technology.

References

[1] *Statcounter FAQ section*. https://statcounter.com/support/faq/.

[2] Bela Amro. “Malware detection techniques for mobile devices”. In: *arXiv preprint arXiv:1801.02837* (2018).

[3] ThreatFabric. *Xenomorph Malware Strikes Again: Over 30+ US Banks Now Targeted*. https://www.threatfabric.com/blogs/xenomorph. Accessed: 2023-09-25.

[4] Zscaler. *Technical Analysis of Anatsa Campaigns: An Android Banking Malware Active in the Google Play Store*. https://www.zscaler.com/blogs/security-research/technical-analysis-anatsa-campaigns-android-banking-malware-active-google. Accessed: 2024-05-27.

[5] E. Flondor, S. Cretu A. Endre, A. Mateescu, and A. Bocereg. “Android SharkBot Droppers on Google Play Underline Platform’s Security Needs”. In: *Bitdefender Labs* (2022). Accessed: 2024-06-08. URL: https://www.bitdefender.com/blog/labs/android-sharkbot-droppers-on-google-play-underlines-platforms-security-needs/.

[6] Lily Hay Newman. “Own an Android? You need to be on the look out for adware”. In: *Wired* (2019). Accessed: 2024-06-08. URL: https://www.wired.com/story/android-adware-malware/.

[7] Alejandro Martín, Raúl Lara-Cabrera, and David Camacho. “Android malware detection through hybrid features fusion and ensemble classifiers: The AndroPyTool framework and the OmniDroid dataset”. In: *Information Fusion* 52 (2019), pp. 128–142.

[8] *Droidbox Github repository*. https://github.com/pjlantz/droidbox/tree/master.

[9] Samrah Mirza, Haider Abbas, Waleed Bin Shahid, Narmeen Shafqat, Mariagrazia Fugini, Zafar Iqbal, and Zia Muhammad. “A malware evasion technique for auditing android anti-malware solutions”. In: *2021 IEEE 30th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*. IEEE. 2021, pp. 125–130.

[10] Ahmad Fikri, Alfan Presekal, Ruki Harwahyu, and Riri Fitri Sari. “Performance comparison of dalvik and ART on different android-based mobile devices”. In: *2018 International Seminar on Research of Information Technology and Intelligent Systems (ISRITI)*. IEEE. 2018, pp. 439–442.

[11] André Danielsson. *Comparing android runtime with native: Fast fourier transform on android*. 2017.

[12] Antonio Ruggia, Dario Nisi, Savino Dambra, Alessio Merlo, Davide Balzarotti, and Simone Aonzo. “Unmasking the Veiled: A Comprehensive Analysis of Android Evasive Malware”. In: *ASIAACS 2024, 19th ACM ASIA Conference on Computer and Communications Security*. 2024.

[13] *Official Android Documentation*. https://developer.android.com/.

[14] Alejandro Guerra-Manzanares, Hayretdin Bahsi, and Marcin Luckner. “Leveraging the first line of defense: A study on the evolution and usage of android security permissions for enhanced android malware detection”. In: *Journal of Computer Virology and Hacking Techniques* 19.1 (2023), pp. 65–96.

[15] Ahmed Hashem El Fiky, Ayman El Shenawy, and Mohamed Ashraf Madkour. “Android malware category and family detection and identification using machine learning”. In: *arXiv preprint arXiv:2107.01927* (2021).

[16] Xiaolu Zhang, Frank Breitinger, Engelbert Luechinger, and Stephen O’Shaughnessy. “Android application forensics: A survey of obfuscation, obfuscation detection and deobfuscation techniques and their impact on investigations”. In: *Forensic Science International: Digital Investigation* 39 (2021), p. 301285.

[17] FastAPI. *FastAPI documentation*. https://fastapi.tiangolo.com/.

[18] Edward Nilsson and Dennis Demir. *Performance comparison of REST vs GraphQL in different web environments: Node. js and Python*. 2023.

[19] Anthony Desnos and Geoffroy Gueguen. “Androguard documentation”. In: *Obtenido de Androguard* (2018).

[20] Juan Lopez, Leonardo Babun, Hidayet Aksu, and A Selcuk Uluagac. “A survey on function and system call hooking approaches”. In: *Journal of Hardware and Systems Security* 1 (2017), pp. 114–136.

[21] Frida. *Frida documentation*. https://frida.re/docs/home/.

[22] Ole André Vadla Ravnås. *Anatomy of a code tracer*. https://medium.com/@oleavr/anatomy-of-a-code-tracer-b081aadb0df8. 2014.

[23] Enrique Soriano-Salvador and Gorka Guardiola-Múzquiz. “Detecting and bypassing frida dynamic function call tracing: exploitation and mitigation”. In: *Journal of Computer Virology and Hacking Techniques* 19.4 (2023), pp. 503–513.

[24] Yehia Ibrahim Alzoubi and Asif Qumer Gill. “An agile enterprise architecture-driven model for geographically distributed agile development”. In: *Transforming Healthcare Through Information Systems: Proceedings of the 24th International Conference on Information Systems Development*. Springer. 2016, pp. 63–77.

[25] Oracle. *Docs*. https://docs.oracle.com/.

DECLARAŢIE DE AUTENTICITATE A LUCRĂRII DE FINALIZARE A STUDIILOR*

Subsemnatul ALBERT ENDRE-LÁSZLÓ ,
legitimat cu CI seria TZ nr. 571221 ,
CNP 5020119350050
autorul lucrării APKANALYZER: ADVANCED INSTRUMENTATION AND MALWARE DETECTION THROUGH DYNAMIC ANALYSIS OF ANDROID APPLICATIONS USING FRIDA
elaborată în vederea susţinerii examenului de finalizare a studiilor de LICENȚĂ organizat de către Facultatea AUTOMATICĂ ȘI CALCULATOARE din cadrul Universităţii Politehnica Timişoara, sesiunea IUNIE a anului universitar 2024, coordonator PROF. UNIV. MIHAI UDRESCU-MILOSAV, luând în considerare art. 34 din *Regulamentul privind organizarea și desfășurarea examenelor de licență/diplomă și disertație*, aprobat prin HS nr. 109/14.05.2020 și cunoscând faptul că în cazul constatării ulterioare a unor declaraţii false, voi suporta sancțiunea administrativă prevăzută de art. 146 din Legea nr. 1/2011 – legea educației naționale și anume anularea diplomei de studii, declar pe proprie răspundere, că:

- această lucrare este rezultatul propriei activităţi intelectuale;
- lucrarea nu conține texte, date sau elemente de grafică din alte lucrări sau din alte surse fără ca acestea să nu fie citate, inclusiv situația în care sursa o reprezintă o altă lucrare/alte lucrări ale subsemnatului;
- sursele bibliografice au fost folosite cu respectarea legislaţiei române şi a convenţiilor internaţionale privind drepturile de autor;
- această lucrare nu a mai fost prezentată în fața unei alte comisii de examen/prezentată public/publicată de licență/diplomă/disertație;
- ~~În elaborarea lucrării am utilizat instrumente specifice inteligenței artificiale (IA) și anume ______ (denumirea) ______ (sursa), pe care le-am citat în conținutul lucrării~~/nu am utilizat instrumente specifice inteligenței artificiale (IA)[1].

Declar că sunt de acord ca lucrarea să fie verificată prin orice modalitate legală pentru confirmarea originalității, consimțind inclusiv la introducerea conținutului său într-o bază de date în acest scop.

Timişoara,

Data 20.06.2024

Semnătura [illegible]

*Declaraţia se completează de student, se semnează olograf de acesta și se inserează în lucrarea de finalizare a studiilor, la sfârșitul lucrării, ca parte integrantă.

[1] Se va păstra una dintre variante: 1 - s-a utilizat IA și se menționează sursa 2 – nu s-a utilizat IA