Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Ina Rekk Bjørnestad

A Hybrid Tool-Chain for Detecting Malware on Android

Master's thesis in Communication Technology
Supervisor: Poul Hegaard
June 2020

NTNU
Kunnskap for en bedre verden

Ina Rekk Bjørnestad

A Hybrid Tool-Chain for Detecting Malware on Android

Master's thesis in Communication Technology
Supervisor: Poul Hegaard
June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

NTNU
Norwegian University of
Science and Technology

Title: A Hybrid Tool-Chain for Detecting Malware on Android
Student: Ina Rekk Bjørnestad

Problem description:

Smartphones are becoming increasingly important in everyday life, such as banking, health, and personal life. However, this also gives rise to new attack vectors for malware writers. After researchers discovered the first malware on Android, the total amount of malware has been increasing steadily. Android has 80% of the market share and is thus the most targeted mobile operating system. The purpose of this project is to investigate and propose a hybrid solution for detecting malware on Android. The hybrid solution will consist of reputation-based security, static- and dynamic analysis. We will incorporate existing tools into our solution and aim at building a fully automated system. The validation of the solution depends on realistic data.

Main contents:

- Investigate and describe background and related work
- Evaluate appropriate tools for static, dynamic, and hybrid systems for detecting malware on Android
- Design (or specify) a pipeline of tools that enables automatic detection of malware on Android
- Implement and validate the system
- Run experiments on real data and discuss the results

Date approved: 12/02/2020
Supervisor: Poul Hegaard, ITEM

Abstract

Mobiles have gained an important role in society and everyday life in recent decades. Today, mobiles are used in several critical functions, such as health, banking, and work tasks. This trend has also meant that malware authors have paid more attention to the mobile platform as a counterpart to the traditional PC. Android is the most popular mobile operating system, with almost 80% of the market. In addition to its popularity, Android also has a far more open system than its most prominent opponent, iOS. The popularity and openness make Android an attractive target for malware authors.

Academic literature has offered numerous solutions to prevent and protect mobile users from harmful products. Besides, the antivirus industry generates billions annually to protect against mobile malware. Nevertheless, new research and solutions are still required to keep pace with the constant development of malware.

The most common methods of analyzing malware are static and dynamic analysis. While static analysis reviews the code without running the application, dynamic analysis inspects the application's behavior. Both methods have their advantages and disadvantages. In this master's thesis, we have used a hybrid detection system, which means that we have combined static and dynamic analysis. The main advantage of such an approach is that we include both methods' benefits and, at the same time, equalize the disadvantages. But this is a very resource-intensive method, which we also had to consider. We have also included reputation-based analysis in our hybrid model, an approach that has not been widely researched. Reputation-based analysis works by gathering and pursuing several attributes such as signature, source, age, and usage statistics. Our hybrid solution consists of several existing tools combined into a pipeline. We selected the tools based on various criteria, and we emphasized, among other things, that the tools should be scalable, automatic, and updated.

We have conducted several experiments with the selected tools. Primarily, we have experimented with both a pre-generated data set and real-time data. Our findings show that including reputation-based analysis can yield better results. In our case, static and dynamic analysis correctly detected 91.58% of our data. By including reputation-based analysis,

98.61% of the data was identified correctly, with a false-positive rate of less than 1%. Our method is fully automated, but not very scalable as it is now. Through the project, we have had a major focus on keeping the solution updated to today's threat landscape. Based on results from experiments with real-time data, we have managed to achieve this goal.

Throughout the work, we observed that there is a need for a universal method for evaluating different detection systems. This is necessary to compare different systems easily. We also noted that there is a need for new tools with open-source code in malware analysis. Both of these observations can be a basis for future work.

Sammendrag

Mobiler har de siste tiårene fått en viktig i rolle i samfunnet og i hverdagslivet. I dag blir mobiler brukt i samtlige kritiske funksjoner, som blant annet helse, bank og arbeidsoppgaver. Denne utviklingen har også medført at skadevareforfattere har viet mer oppmerksomhet til den mobile plattformen som et motstykke til den tradisjonelle PCen. Android er det mest populære mobile operativsystemet med nesten 80% av markedet. I tillegg til populariteten, har Android også et langt mer åpent system enn dens største motstander, iOS. Dette gjør Android til et attraktivt mål for skadevareforfattere.

Den akademiske litteraturen har foreslått utallige løsninger for å hindre og beskytte mobilbrukere fra skadevare. Dessuten genererer antivirusindustrien milliarder årlig for å tilby beskyttelse mot skadevare på mobil. Likevel kreves det stadig ny forskning og løsninger for å holde tritt med skadevarens konstante utvikling.

De mest vanlige metodene for å analysere skadevare er statisk- og dynamisk analyse. Statisk analyse gjennomgår koden uten å kjøre applikasjonen, mens dynamisk analyse inspiserer applikasjonens oppførsel. Begge metodene har sine fordeler og ulemper. I denne masteroppgaven har vi brukt et hybrid detekteringssystem, som innebærer at vi har kombinert statisk- og dynamisk analyse. Den største fordelen ved en slik tilnærming er at vi inkluderer fordelene ved begge metodene og samtidig utjevner ulempene. Men dette en svært ressurskrevende metode, noe som vi også har måttet vurdere. Vi har også inkludert rykte-basert analyse i vår hybride modell, en metode som ikke er særlig brukt i den akademiske sirkel. Rykte-basert analyse fungerer ved å samle og følge flere attributter som signatur, kilde, alder og bruksstatistikk. Vår hybride løsning består av flere eksisterende verktøy. Disse ble valgt ut i fra en rekke kriterier, og vi la blant annet vekt på at verktøyene skulle være skalerbar, automatisk og oppdaterte.

Vi har gjennomført flere eksperimenter med de valgte verktøyene. I hovedsak har vi eksperimentert med både et pre-generert datasett og sanntidsdata. Våre funn viser at en rykte-basert analyse potensielt kan erstatte den tradisjonelle signaturtilnærmingen. Dessuten viser våre funn at å inkludere rykte-basert analyse kan gi bedre resultater. I vårt tilfelle

detekterte statisk- og dynamisk analyse 91.58% av dataen vår riktig. Ved å inkludere rykte-basert analyse, ble 98.61% av dataen detektert riktig, med en falsk-positiv rate på under 1%. Vår metode er helt automatisert, men ikke særlig skalerbar slik den er nå. Gjennom prosjektet har vi hatt et stort fokus på at løsningen skal være oppdatert til dagens trusselbilde. Ut i fra resultater på sanntidsdata har vi klart å nå dette målet.

I løpet av arbeidet observerte vi at det er et behov for en universell metode for å evaluere ulike detekteringssystem. Dette er nødvendig for å enkelt kunne sammenligne ulike systemer. Vi observerte også at det er et behov for nye verktøy med åpen kildekode innen skadevareanalyse. Begge disse observasjonene kan være et grunnlag for fremtidig arbeid.

Preface

This thesis was prepared during the spring of 2020 at the Norwegian Institute of Science and Technology, Faculty of Information Technology and Electrical Engineering, Department of Communication Technology and Information Security. The thesis was conducted in collaboration with Norton LifeLock.

I would like to thank my supervisor, Poul Hegaard, at the Norwegian University of Science and Technology. Poul is a friend and a mentor, and he is always willing to help. I am very grateful for his excellent advice and counseling.

I would also like to thank the professionals, Felix Leder and Michal Zuberek, from Norton LifeLock for their expert opinions.

Finally, I want to thank my family and friends. Special gratitude to my fellow student Ingvild Løes Nilsson for a second opinion on the master thesis.

Contents

List of Figures

List of Tables

Chapter 1
Introduction

This chapter presents the motivation behind the master thesis, outlines the purpose of the project, and present the research questions that we aim to answer throughout the thesis. We give the remaining structure of the thesis at the end of the chapter.

Keywords: Android malware, static analysis, dynamic analysis, reputation-based analysis, hybrid analysis, tools

1.1 Motivation and justifications

According to DataReportal [Dat19], there are 5.11 billion unique mobile users in the world today, with a 2% increase in the past year. As a percentage of the total population, 52% are mobile internet users. Additionally, smartphones are becoming increasingly important in everyday life, such as banking, health, and personal life. This mobile evolution gives rise to new attack vectors for malware writers supplementary to traditional computer malware.

The first reported computer malware occurred in 1981-1982 and targeted games on Apple II [SZ03]. The computer malware was already evolving and persistent when the first mobile malware named Carbir occurred in 2004, targeting the Symbian operating system [Car19]. Android was launched in 2008, and its popularity has been exploding ever since. Today, Android has almost 80% of the market share and is thus the most targeted mobile operating system for malware writers [Sta19].

After researchers discovered the first malware on Android in 2010 [Ken15], the total amount of malware has been increasing rapidly. As of May 2019, the total number of Android malware detections amounted to over 10.5 million programs [Cle19b]. Equally important, malware has gotten extremely sophisticated. Academics and

industries have launched and proposed abundant solutions to enable detection of malware on Android. The anti-virus industry has generated billions of dollars in the past decade by commencing protection against malware. However, with a continually changing threat landscape, detection systems must regularly evolve and update to reflect the recent changes.

The most common detection techniques for malware are static- and dynamic analysis. While static analysis examines an application's code without running it, dynamic analysis executes the app and observes its behavior. The purpose of this thesis is to investigate and propose a hybrid solution for detecting malware on Android. The hybrid solution consists of reputation-based analysis, static- and dynamic analysis. A reputation-based analysis is based on an application's intrinsically collected reputation. It functions by gathering and pursuing several attributes such as signature, source, age, and usage statistics. Although reputation-based analysis is commonly used among anti-virus providers [ZRN10], it is rarely used in the research industry. To the best of our knowledge, no public research project has incorporated reputation-based analysis in their hybrid malware detection system.

As reported by Statista, an average of 6,140 mobile apps were released through Android's official market place Google Play Store every day, measured from the third quarter of 2016 to the first quarter of 2018 [Cle19a]. In addition to those applications released on Android's official market place, many third-party markets exist, for example, anzhi and appchina. Because of this enormous volume of applications, we aim at building a fully automated up-to-date system. To do so, we integrate existing tools into a pipeline. The pipeline is evaluated with realistic data.

Specialization project

During the specialization project fall 2019, we researched malware analysis on both Android and iOS. Other mobile operating systems, such as Windows, have a small fragment of the market share and was therefore not considered. Malware rarely emerges on iOS, especially compared to Android. Conducting malware analysis on iOS is a very cumbersome task, as iOS is a far more restricted and closed system than Android. Thus, Android became the operating system of choice for the master thesis. We conducted a literature review on Android malware analysis and found that there is a need for more recent data samples to evaluate malware detection solutions. Current research projects tend to use old and outdated data that do not reflect the current situation. We also found that hybrid solutions are the most promising option for malware analysis, as it can combine benefits and minimize the drawbacks of the most common malware analysis techniques. Therefore, it was decided to implement

a hybrid detection system.

1.2 Research questions

The overwhelming number of malware on Android poses a threat to the user's mobile security. Even though abundant research projects exist, the protection and detection against Android malware have to continually improve and evolve to keep track of the ever-evolving threat landscape. The goal with this thesis is to solve a part of the puzzle, and hopefully improve the future detection of malware. To do so, we investigate the following research questions:

RQ1: What should be the trade-off between static- and dynamic analysis to identify mobile malware efficiently?
Determining appropriate measures to decide the trade-off between reputation-based, static, and dynamic analysis is a subtask.

RQ2: How can the selected tools be incorporated into a toolchain, and where is reputation-based analysis useful?

RQ3: To what extent does the effectiveness of the solution differ on old files compared to recent files?

RQ4: How can the performance of hybrid analysis systems be improved by combining it with results from reputation-based analysis?

1.3 Thesis outline

- **Chapter 2: Background** describes relevant concepts and background information for the remaining thesis. The chapter includes Android operating system fundamentals, aspects from Android's security model, malware characterization and evolution, and malware detection techniques.
- **Chapter 3: Related Work** gives a throughout examination of related research in the area.
- **Chapter 4: Methodology** explains the methods that we used to collect data, select tools, build our pipeline, and evaluate the results.
- **Chapter 5: Experiments and Tools Selection** presents the tested tools and discuss their advantages and limitations. The chapter also explains how experiments were conducted on the proposed tool-chain.

- **Chapter 6: Results and Discussion** presents the results obtained from the developed pipeline of tools. We discuss the results in terms of selected evaluation metrics, requirements, and research questions.
- **Chapter 7: Conclusion and Future Work** concludes and summarizes the thesis. In the end, future work is suggested based on the findings and limitations.

Chapter 2
Background

This chapter presents a brief but essential background on Android Operating System (OS) fundamentals and presents relevant concepts from its security model. Then, a characterization of malware and distribution on Android is conferred. Finally, the different techniques for malware analysis are explained. *Android Security Internals* [Ele14], and *Android Malware Analysis* [Ken15] are used throughout the chapter for relevant background information, unless other sources are given. Interested readers are encouraged to access these books for more details.

2.1 Android Operating System fundamentals

Android is a mobile operating system developed by Google and was first released in 2008. As can be observed from Figure 2.1, Android is based on the *Linux kernel*. The Linux kernel provides drivers for hardware, networking, file-system access, and process management. Traditional OSs divides into two spaces: user space and kernel space. User-space is where the user processes and applications persist, and kernel space is where the actual operating system lives.

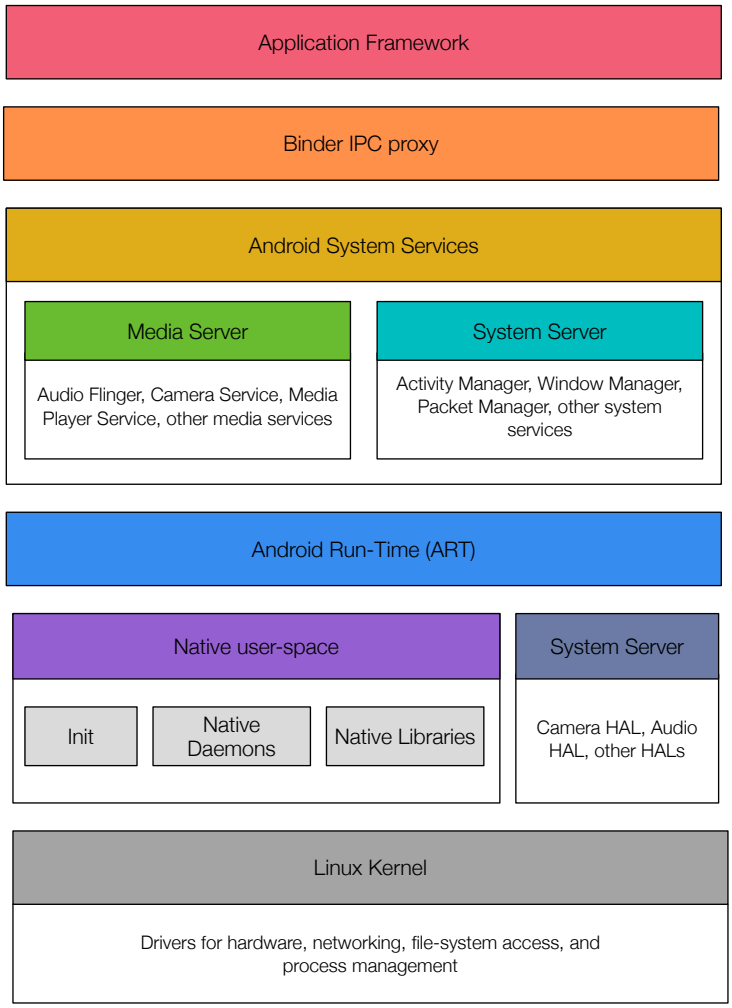


Figure 2.1: Android's architecture.

The *native user-space* remains on top of the Linux kernel. It consists of the *Init* binary, which is the first process started and starts all other processes. The native user-space also consists of native daemons and a few hundreds of native libraries. *Hardware abstraction layer (HAL)* also persist above the Linux kernel, and defines a standard interface for hardware vendors to implement. HAL enables Android to be agnostic about lower-level driver implementations [HAL20].

Android Runtime (ART) is an application runtime environment used explicitly for Android. Dalvik was the original virtual machine used by Android, but ART replaced

it in Android 5.0 "Lollipop." In replacing Dalvik, ART translates the application's byte code into native instructions. The device's runtime environment later executes these instructions.

The components described until now are the fundamental building blocks necessary to implement *System Services*. System Services make up all the essential features on Android, including network connection, activities, display, and touchscreen support. Like other Unix-like systems, a process cannot access another process's memory. However, the kernel has control over all processes and therefore, can expose an interface that enables *Inter-process communication (IPC)*. IPC occurs through *Binder*. An Android process can call a routine in another Android process using Binder to identify the method to invoke and pass the arguments between processes, as shown in Figure 2.2.

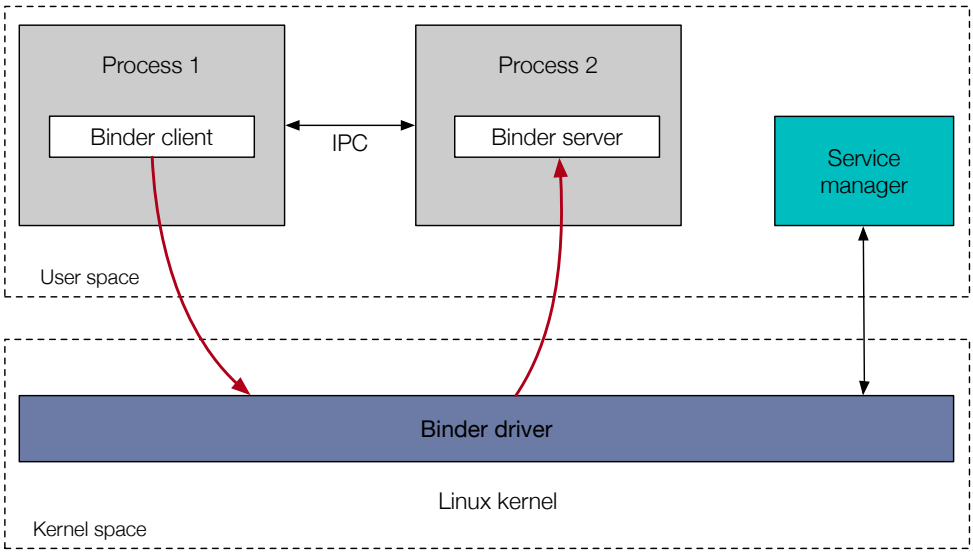


Figure 2.2: Binder allows for Inter-Process Communication (IPC).

Higher level components such as *Intents, Messengers* and *ContentProviders* are built on top of Binder.

Applications can be either user-installed (from a third-party application store) or system applications. System apps are included in the OS image and cannot be modified or deleted by a user. Therefore, these apps are considered more secure and

thus given more privileges than user-installed apps. User-installed apps, on the other hand, can be changed or uninstalled by the user. Each user-installed app lives in a dedicated security sandbox to prevent them from affecting other apps.

The main application components are Activities, Content Providers, Broadcast Receivers, and Services. *Activities* are the main building blocks of Android GUI applications. An application can have multiple activities, which can start independently. If allowed, another application can trigger the activities to start. *Services* runs in the background without user interaction. They are generally used to perform some long-running operations such as file downloads. *Content providers* use IPC to provide an interface to the application data. Content providers are generally used to share an application's data with another application. *Broadcast receivers* responds to system-wide events, named broadcasts. They originate from either an application or from the system itself.

2.2 Android Security Model

Android is a privilege separated operating system, where each application has its own Linux user ID (Linux UID) and persists in a security sandbox. The sandbox provides the system with standard Linux process isolation and restricted file-system permissions. Higher-level system functions are implemented as a set of collaborating system services that are communicating through Binder.

Permissions

By default, applications have very few privileges and thus must request permission to interact with system services (for example, camera and internet), access sensitive user data (for instance, contacts and SMSs), or communicating with other apps. Depending on the feature, the system might grant permissions automatically or might prompt the user to approve the request. Permissions divide into three security levels:

1. **Normal permissions** are granted automatically at install time. These covers areas were there is a little security risk, like setting the timezone.

2. **Dangerous permissions** needs to be explicitly granted by the user, and cover areas where the app wants data or resources that involve the user's private information, or could potentially affect the user's stored data or the operation of other apps.

3. **Signature permissions** are granted at install time, but only when the same certificate signs the app that attempts to use permission as the app that defines the permission.

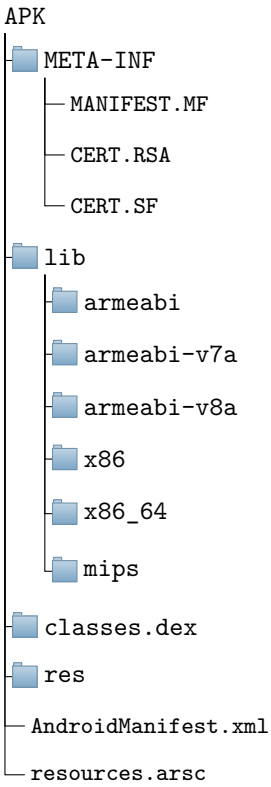
There are a few permissions, such as SYSTEM_ALERT_WINDOW and WRITE_SETTINGS, which do not behave as the permissions described above. These are called special permissions and are particularly sensitive. Permissions are defined in the application's manifest and granted at install time. The system uses the UID to determine which permissions the application has been granted and enforce them at runtime.

Key signing

All apps, which run on Android devices, have to be signed with a certificate identifying the author of the app. Nonetheless, Android allows applications to be self-signed, which has a fundamental flaw. All Android applications and updates must include a digital signature. However, vendors are allowed to use the same certificate in different applications. Besides, the application updates can be signed with the same certificate. Therefore a certificate does not uniquely identify any given application. Thus, an Android developer can claim to be anyone they want to be, including Bank of America, for instance. Compared to iOS, this would be a much harder achievement, as Apple does not allow for self-signed certificates.

2.3 Android Application Package

Android Application Package (APK) is the distribution format used for Android to distribute and install applications. The files are a type of archive file based on the JAR file format with the extension .apk. They typically contain the following:



MANIFEST.MF is the manifest file containing metadata about the application. *CERT.RSA* is the certificate of the application. *CERT.SF* holds the list of resources and an SHA-1 digest of the corresponding lines in the MANIFEST.MF file. *lib* is the directory containing the compiled code dependent on the library. It is divided into several directories, where each directory is compiled code for the specific processor. *AndroidManifest.xml* is an additional manifest file that contains information about intents, names, versions, external reference libraries, and permissions. It is frequently used within static malware analysis to extract permissions, intents, or others. *classes.dex* contains the classes compiled in .dex format, understandable by the Dalvik Virtual Machine and Android runtime. The *res* folder consists of the resources in the application that is not compiled into *resources.arsc*. resources.arsc are the precompiled resources.

2.4 Characterizing malware on Android

2.4.1 Taxonomy

Malware (**Mal**icious soft**ware**) is explained and defined by several researchers. Other terms used as an alternative to malware include malicious code, malicious software, or malcode. The original definition of malware in [Fre84] was: "*a program that can infect other programs by modifying them to include a possible copy of itself,*" but was later updated [DMAS16]. Its objective has later explained malware: malware intends to be destructive [CJS+05] or cause sabotage to the system [MM00].

Malware is also commonly used to refer to a group of malicious software: Viruses, Worms, and Trojans. A virus is a self-replicating program. It needs to attach itself to a host program to be able to reproduce itself. A worm is also a self-replicating program, but indifference from a virus, a worm does not need to attach itself to a host program. Besides, worms tend to repeat itself over network connections, while viruses replicate on the host computer. A trojan is a program that mimics a legitimate program. Without user awareness, the trojan may perform unwanted actions [Kar03].

Further, the antivirus industry frequently uses the term "Potentially Unwanted Applications (PUAs)". The term is used to describe software that is located somewhere between malware and benign applications. For this reason, PUAs are also referred to as "grayware" [SMJ16]. PUAs includes applications such as adware and spyware. Adware is primarily aiming to make money through advertising. It may collect personal information about users for this purpose. Mobile adware is particularly aggressive and can gather sensitive information such as geo-location, explode the device with advertisements, or be costly in terms of data usage and messaging charges. Spyware intends to track and records the user's behavior and send it to a third party. It commonly gathers sensitive information without the user's knowledge. Similar to malware, PUAs can consume large amounts of memory, CPU, or other resources, causing a performance degradation of the device [SMJ16].

Another term from the gray zone is "Riskware." Riskware is legitimate apps that *can* be exploited by a person with malicious intent [Rey20]. They are not designed with malicious intent but contain weaknesses that can be exploited. There is a fine-grained line of whether or not we should include such applications in a definition of malware, and application within the category should be considered from its context.

Throughout the remaining parts of this project, we use the term malware for any

software with destructive or malicious intent. Defining malware in terms of its intent is the most describing one, and captures the different types of malware, including those in the gray zone. Note that we could have divided it into more fine-grained terms, but for simplicity, we use malware unless otherwise stated.

2.4.2 Evolution

Android's popularity makes it the preferred target for malware. Since Android was released in 2008, its popularity has been expanding significantly, as can be seen in Figure 2.3. Apart from the popularity, Android is open-source, making it an easier target for malware. Besides, Android suffers from the fragmentation problem: unlike iOS, the Android operating system can be implemented on hardware provided by different vendors. Although the fragmentation has contributed to its popularity, it also gives rise to security flaws. Vendors might provide a different level of security, and the security updates are received at various times. Unofficial app stores also exist. In general, these have a higher malware infection rate than the official play store [TFA+17]. Some countries do not have access to the official app store and are thus forced to use the unofficial stores.

In August 2010, the first Android Trojans, FakePlayer and DroidSMS, were discovered in the wild [Ken15]. The former was a Trojan horse that attempted to send premium-rate SMSs to a hard-coded phone number. Similarly, DroidSMS was a classic SMS fraud app that sent SMSs to premium rate phone numbers. Since then, the amount of mobile malware has been exploding, and Kaspersky reported a doubling of mobile malware in 2018 [Kas19]. Malware has not only been growing in terms of amount, but it has also gotten more sophisticated within the years. A recent malware named "Agent Smith" managed to infect around 25 million devices worldwide [Sip20]. It was disguised as a Google-related application that users were willing to install. Without the user knowing, Agent Smith was able to replace legit apps that the user had installed with malicious versions. Agent Smith is only one example of recent malware, but it shows how sophisticated and intelligent mobile malware may be at this point.

Malware development can be very lucrative, and a report from 2013 showed that malware writers could earn up to 12,000 USD per month [TFA+17]. An increase in black markets[1] have also provided more incentives for profitable malware writing.

[1]Black markets are markets were stolen information, system vulnerabilities, malware source code, malware development tools, and similar are sold.

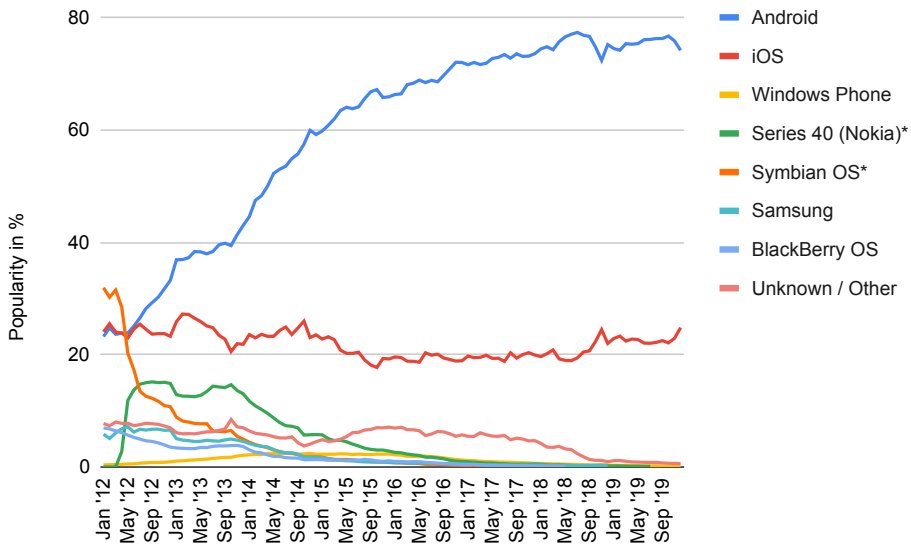


Figure 2.3: Market shares for mobile operating systems from January 2012 - November 2019. The data was collected from StatCounter [Sta19].

2.4.3 Current threat landscape

The current threat landscape of Android is enormous and hard to define correctly. To get an idea of the current situation, we use public reports published by antivirus companies.

According to McAfee's mobile threat report for the first quarter of 2019, "FakeApp" was one of the greatest threats of 2018 [SD19]. FakeApp is a family of malware that mimics real applications, such as the game Fortnite, and can appear to be very real. In the same report, McAfee announced that mobile banking trojans and spyware is one the raise. Spyware, sometimes called stalkerware, was also reported as a growing threat by MalwareBytes [Mal20] and Kaspersky [Che20]. A threat that has been growing significantly in 2019 is adware [Che20]. Kaspersky Lab disclosed that four out of ten places in their top 10 mobile threats list was reserved for adware. The main increase was due to the sharp growth in the "HiddenAdd" family. The same family was also the second most detected malware family by MalwareBytes in 2019 [Mal20]. The HiddenAdd family vitally displays ads in a very aggressive way. In addition to adware, Kaspersky saw an increase in riskware, specifically in risk tools.

In this category, the sharp escalation came from the family SMSreg. The SMSreg family is primarily characterized by money transfers via SMS, which the user is not explicitly informed of.

Figure 2.4 demonstrates the current situation of mobile threats, reported by Kaspersky. As can be observed, there was a considerable increase in adware from late 2019 to the first part of 2020. In general, the percentage of mobile trojans decreased from 2019 to 2020.

2.4.4 Distribution of malware

Repackaging

Repackaging is one of the most common ways to distribute malware [ZJ12]. Attackers piggyback their malicious payload in a popular application and redistribute it. Generally, they download an attractive application, disassemble it, and re-assemble it with a malicious payload enclosed. Finally, they distribute the new, malicious version on Google Play or other third-party stores.

Update attack

Indifference from repackaging, an update attack distributes the malicious payload *dynamically*. In an update attack, the attacker might still repackage popular applications. However, instead of including the malicious payload as a whole, they include an update component that downloads or fetch the malicious payload at run-time. Because this happens dynamically, a static analysis might be evaded and fail to catch the malicious payload.

Drive-by download

In this context, a drive-by download happens without a user's knowledge. It may occur when a user visits a web site, opens an e-mail attachment, or is lured into clicking a link. Note that this attack is not specific to mobile platforms, but are primarily attracting users into downloading "valuable" and "impressive" apps.

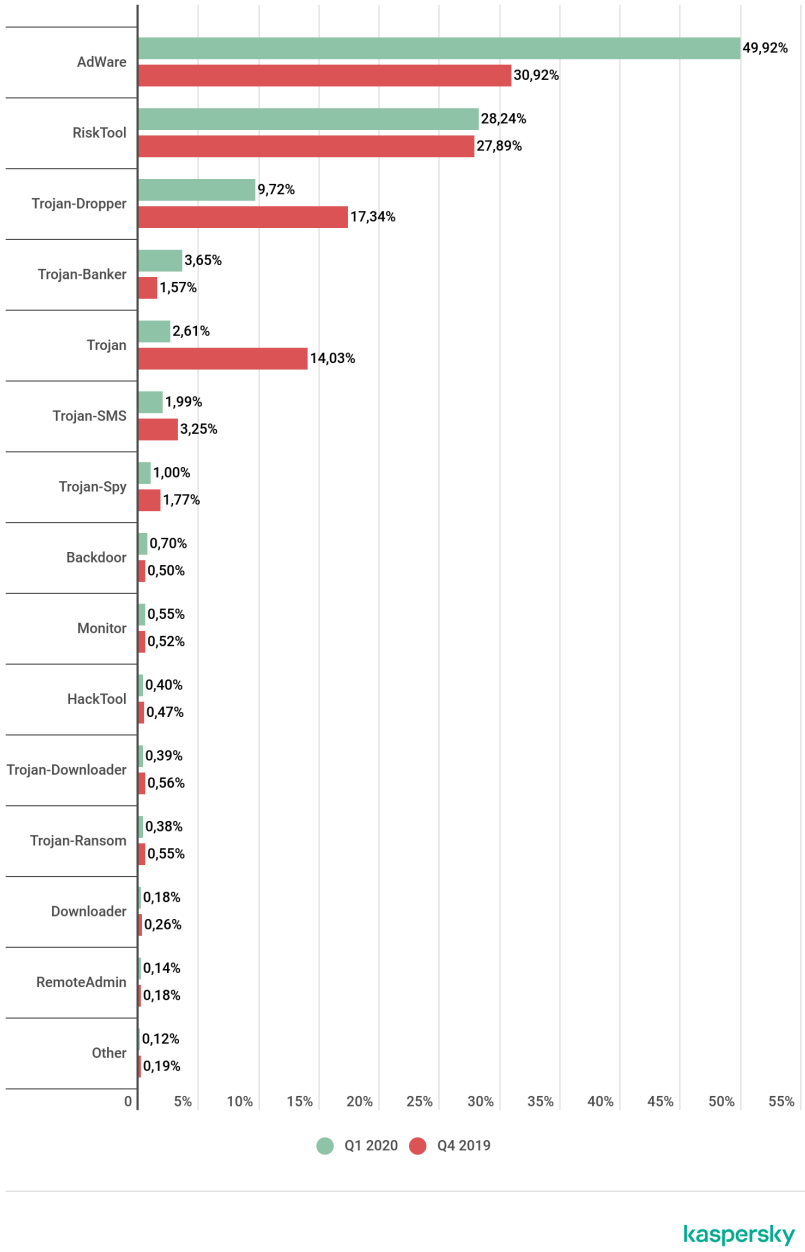


Figure 2.4: Mobile threats in the quarter of 2019 compared to the third quarter of 2020, detected by the antivirus company Kaspersky [Che20].

2.4.5 Market places

Google Play

Google Play[2] is the official app store for Android apps. While the majority of the malware persists at third-party app stores, many have found their way into Google Play. When Google is alerted by crook apps, they take the apps down, but not always as fast as necessary. For example, in early 2011, Google pulled more than 50 apps infected with DroidDream but chose not to trigger automatic uninstalls from users' devices [MD12].

Third party app stores

Third-party app stores are unofficial market places. They are less restricted than Google Play, so developers can tweak apps as they want. They are often used to offer hacked versions of a paid app. Nonetheless, fewer restrictions also make it easier for malware writers to distribute their malicious code.

2.5 Malware detection techniques

Static analysis and dynamic analysis are the main methods for detecting malware. Static analysis examines artifacts in the executable without running it, while dynamic analysis monitors the runtime behavior of the executable. Both methods have their advantages and disadvantages.

2.5.1 Static analysis

Static analysis examines applications without running them. One of the most significant advantages of this method is the possibility of obtaining full code coverage. Static analysis is also the most flexible part of malware analysis, as it can be conducted from a multitude of operating systems. The process of static analysis of Android malware is similar to traditional Windows or Linux malware. The difference between, for instance, Windows and Android are how APKs are packaged and compiled compared to a Windows binary. Windows binaries are compiled to executables with an MZ header. Android applications are compiled as an APK that can be unpacked into separate files, including the source code, a manifest, and other files. Typically, static analysis collects file hashes, metadata, and strings. Commonly collected static features specific to Android include permissions, services, providers, receivers, activities, Application Programming Interface (API) calls, and others. A challenge

[2]https://www.play.google.com

Table 2.1: The most frequent approaches within static analysis.

Method	Advantages	Limitations
Signature-based detection	Efficient against known malware.	Fails to detect new malware. Code obfuscation may circumvent the detection method.
Permission-based detection	Only one file is scanned.	Limited difference between malware and benign.
Bytecode-based analysis	Can capture behavior.	More resource-intensive.

with static analysis is that the code might be obfuscated or encrypted, making the analysis troublesome. Code obfuscation is the process of hiding application logic during compilation so that the logic of an application is difficult to follow. It is commonly applied by industries to protect intellectual property, but also by malware writers to evade detection. The static analysis does not inherently capture the full runtime behavior, in which dynamic analysis must be applied.

The static analysis further divides into three commonly used methods for malware detection: signature-based detection, permission-based detection, and bytecode based detection [ASKA16][GO18]. In a *signature based detection* method, patterns are extracted from the code to create unique signatures. The app is categorized as malware if the extracted signatures match one of a known malware family. Code obfuscation is an obvious drawback of this method as it can circumvent the detection. Another drawback is the inability to detect unknown malware ("zero-days"), although the method is very efficient against known malware. *Permission-based detection* examines the permissions in the application and search for any anomalies. It is an easy way to scan the application as it only scans the application's manifest. However, there is a minimal difference between malicious and benign in terms of permissions requested by applications[ASKA16][GO18]. *Bytecode-based analysis* facilitates the recognition of the application's behavior. Control and data flow analysis can help detect suspicious functionalities performed by the application. Nevertheless, since this analysis method operates at the instruction level, it is also the most resource-consuming method in terms of power and memory.

Open source tools

Academics and industries have proposed various tools for static analysis. Unfortunately, only a few of them are available to the public. Within the tools available,

ApkTool[3] is frequently used to decompile an application executable. It enables reading the Android Manifest and the smali, extracted from Dalvik Bytecode. *dex2jar*[4] is used to decompile the dex source files to JAR, which can be read later by the jd-gui. *jadx*[5] converts the dex files to java files. If the conversion works properly, it enables reading the original code as java. *AndroGuard*[6] is a Python library that can extract various features from an APK file by applying static analysis. Other open-source tools widely leverage AndroGuard. Among others, MobSF, AndroPyTool [MGLCC18], VirusTotal, and CuckooDroid use AndroGuard. More specified tools also exist. For instance, *FlowDroid* [ARF+14] is a tool that runs taint analysis to follow the information flow.

Online scanners

Online scanners also exist, to let users scan a suspicious file. Unfortunately, online scanners are often very limited in terms of allowed file sizes, long processing times, and freedom of choice. Available static scanners for Android files are AVCUndroid[7], Dr. Web Online[8], VirScan[9], and Kaspersky[10].

2.5.2 Dynamic analysis

Indifference from static analysis, dynamic analysis observes the behavior of the application when it is executed and can identify malicious behavior in the runtime environment. Dynamic analysis can include observing system calls, tracking data stream, tracing directions, and detaching function parameters [GO18]. It is frequently executed within a *sandbox* - a mechanism for separating running programs, or on a real device. Although dynamic analysis captures the application's behavior better, it cannot capture full code coverage, and it is far more resource-intensive than static analysis.

Anomaly-based detection, taint analysis, and emulation based detection are the most frequently used techniques within dynamic analysis [ASKA16][GO18]. *Anomaly based detection* is the process of comparing definitions of what activity is considered normal against observed events to identify significant deviations. Indifference from a signature-based approach, this method can detect unknown malware. However,

[3] https://ibotpeaches.github.io/Apktool/
[4] https://github.com/pxb1988/dex2jar
[5] https://github.com/skylot/jadx
[6] https://github.com/androguard/androguard
[7] https://undroid.av-comparatives.org/
[8] https://vms.drweb.com/online/
[9] https://www.virscan.org/
[10] https://www.kaspersky.no/

Table 2.2: Commonly used techniques within dynamic analysis.

Method	Advantages	Limitations
Anomaly-based detection	Can detect unknown malware.	Erroneous when benign apps show the same behaviour as malware.
Taint analysis	Capture the information flow.	Not suitable for real-time analysis.
Emulation-based detection	Less resource intensive than a sandbox.	Resource intensive compared to other approaches.

the method fails when benign apps behave similarly to what is defined as malicious behavior, for example, more API invocations or higher resource consumption. *Taint analysis* is a type of information flow tracking which, for instance, can track sources of sensitive information. This method is not suitable for real-time analysis and downgrades performance significantly [ASKA16]. *Emulation based detection* executes within an emulator, which is more lightweight compared to a full-featured sandbox. An emulator only emulates the execution of the sample itself. It creates temporary objects that interact with the samples.

Sandbox Security

A sandbox is an isolated environment that imitates end-user operating environments. It is used to run potentially harmful samples in a safe environment, without risking damage to the host or network. Thus, the dynamic analysis of malware is frequently performed within a sandbox. However, malware writers are always working to evade detection, and several evasion techniques exist. One common technique is to detect the sandbox. Because the sandbox is somewhat different from a real host environment, malware can detect the differences and terminate immediately or stall the execution of malicious activities. Malware can also take advantage of the sandbox by implementing context-aware triggers. For instance, by implementing logic bombs, the malware writer can delay code execution for a certain period or until a particular event is triggered. Another common trick is to exploit the sandbox's weaknesses and gaps, for example, by using obscure file-formats or huge files that the sandbox cannot process.

Open source tools

As with static analysis, only a few existing tools are publicly available. *DroidBox*[11] is one such tool, which connects to an Android emulator to perform dynamic analysis. Unfortunately, the last update was received four years ago. Consequently, the highest SDK version it supports is version 16.

Online sandboxes

Online sandboxes let the user scan and execute a doubtful file in a safe environment. Accessible online sandboxes for Android are Joe Sandbox[12] and AMAaaS[13].

2.5.3 Hybrid analysis

Hybrid analysis means that both static- and dynamic analysis is combined. The apparent benefit of such an approach is that the advantages of both analysis types are included. However, this type of approach is also the most resource-intensive.

Open source tools

AndroPyTool [MGLCC18] is one of the more recent advanced tools. It combines AndroGuard, FlowDroid, DroidBox, AVClass, VT, or Strace to perform advanced static and dynamic analysis. *MobSF* [Abr16] is a tool for mobile penetration testing and malware analysis. It is a fully automated, cross-platform tool that works on Android, Windows, and iOS. MobSF uses AndroGuard for static analysis, and the dynamic analysis is performed in an Android VM named Genymotion. In the dynamic analysis part, it provides support for dynamic instrumentation using Frida[14]. *CuckooDroid*[15] is an extension of the Cuckoo Sandbox[16] that enables automated hybrid analysis. However, it is not maintained anymore, with the last update 3 years ago. Currently, it only supports Android 4.1, which is too old to reflect recent changes.

2.5.4 Reputation-based analysis

Reputation-based analysis is a mechanism typically implemented in anti-virus engines, such as in Norton's security products [ZRN10]. In a reputation-based security

[11] https://github.com/pjlantz/droidbox
[12] https://www.joesecurity.org/
[13] https://amaaas.com/
[14] https://www.frida.re/
[15] https://github.com/idanr1986/cuckoo-droid
[16] https://cuckoo.readthedocs.io/en/latest/

system, an executable file or an application is classified as unsafe or safe based on its intrinsically collected reputation. It enables predictable file safety, depending on its overall use and reputation from a vast user community. Reputation-based analysis works by gathering various file attributes, such as signature, age, source, number of downloads, and global usage statistics. It can also consider the vendor's reputation [TBA12]. Applications from Google's official application store will, for example, have a much better reputation than an application downloaded from an unknown, third party application store. A reputation engine typically analyzes the gathered data by using statistical analysis. This approach works very well for prevalent malware and is the least resource-consuming method.

2.6 Existing datasets

Generally, pre-generated datasets are used to evaluate malware detection systems. Pre-generated datasets are often well-labeled and structured, making the process of evaluation less cumbersome. A summary of the available datasets for Android malware is shown in Table 2.3.

The most popular pre-generated datasets in literature are the Android Malware Genome Project [ZJ12] and Drebin [ASH+14]. MalGenome was the most studied and well-labeled dataset for an extended period but was discontinued in 2015 due to resource limitations. The data samples in Drebin were collected from 2010 until 2012. The Contagio Minidump[17] is a smaller dataset observed in the literature, and consists of 189 malware samples seen in the wild. It was downloaded on October 26th, 2011. Wei et al. [WLR+17] discuss the need for more up to date datasets and the need for more trustworthy, complete information. Therefore, they constructed and published a more reliable, recent dataset named AMD. AMD consists of 24,650 labeled Android malware samples that are classified in 135 varieties within 71 families, whose discovery dates range from 2010 to 2016. AMD is still the most recent public pre-generated dataset containing general Android Malware.

Datasets containing both malware samples and benign apps also exist. Li et al. present AndroZoo [Li,17], a growing collection of Android applications. The applications originate from various sources, including GooglePlay and AppChina, among others. It currently contains 10,577,653 different APKs, including both malware and benign apps, but are not labeled. Android Adware and General Malware (AAGM) [LAG+17] is another mixed data set containing 1900 (1500 benign and 400 malware) applications

[17] http://cgi.cs.indiana.edu/ nhusted/dokuwiki/doku.php?id=datasets

Table 2.3: Summary of available malware datasets.

Database	Variety	Time interval
Android Malware Genome Project	1,200 Malware samples	2010 - 2011
Drebin	5,560 Malware samples	2010 - 2012
AMD	24,553 Malware samples	2010 - 2016
AAGM Dataset	1900 Mixed samples	2008 - 2016
AndroZoo	10,165,192 Mixed samples	2010 –
Kharon	7 Malware samples totally reversed and documented	2011 - 2015
Android ProGuard Dataset	10479 Obfuscated malware samples	2011 - 2015
The Contagio Minidump	189 Malware samples	2011
UpDroid	2,479 Malware samples	2015 - 2019

from a selected variety of adware and general malware families. The AAGM dataset is captured by installing Android apps on real smartphones semi-automated.

More specialized datasets include Kharon [KLLT16], UpDroid [AS18], and the Android ProGuard Dataset [MAC+15]. Kharon is a small dataset containing 7 malware samples that are completely reversed and documented. The dataset was constructed to help researchers evaluate their work. UpDroid is specialized in the update technique; it consists of 2,479 malware samples that use the update technique to evade detection. The Android ProGuard Dataset contains 10479 samples, obtained by obfuscating the MalGenome and the Contagio Minidump datasets with seven different obfuscation techniques.

Chapter 3
Related Work

Android has almost 80% of the global market share today [Sta19]. Within the last years, Android has increased its features with more advanced ones, for example, more health features, mobile banking, and mobile wallet. As the features are becoming more sophisticated, they are becoming a more exciting target for malware writers. Consequently, the total number of malware has been steadily increasing in recent years, according to Kaspersky [Kas19]. As expected, malware researchers across industries and academics have put enormous effort into designing novel solutions to detect different kinds of malware. This chapter will present some of the work done in the area. The related work is further divided into three categories: static approaches, dynamic approaches, and hybrid approaches. We focus on extracting relevant information related to building a hybrid detection system for Android malware. In particular, we examine how the static- and dynamic analysis was performed in terms of extracted features and tools. We also observe where the data was collected, how well the proposed solution performs, and potential limitations.

3.1 Static approaches

Arp et al. [ASH+14] presents a lightweight system that utilizes static analysis and machine learning, named Drebin. Drebin gathers features from the application's manifest and code to perform deep static analysis. Features include hardware components, restricted- and suspicious API calls, network addresses, app components, and used- and requested permissions. They then combine the extracted features into a joint vectored space, where patterns and combinations are analyzed geometrically. They employ 23,453 applications and 5,560 malware samples in their evaluation. The authors collect their samples from various sources in the range 2010 - 2012, including Google PlayStore, numerous Chinese and Russian markets, malware forums,

security blogs, and Genome. The detection rate was measured to be 93%, with 1% false positives on average. Nonetheless, their approach cannot disclose samples from unknown malware families spontaneously but is dependent on several files to learn their nature.

Fereidooni et al. [FCYS16] propose Anastasia, a system for detecting Android malware. They develop a tool named uniPDroid to extract features from applications, including intents, permissions, malicious activities, and system commands. They conduct the classification of malware and benign apps by utilizing different machine learning techniques. Their dataset consists of well-labeled applications collected from Genome, Drebin, M0droid, and VirusTotal (2009 - 2015). Finally, their detection system achieves an accuracy of 97% with a 5% false-positive ratio and a 2.7% false-negative ratio. As for future work, they suggest extracting more features from applications, such as memory and CPU consumption, Inter-process communications, and system calls.

Arzt et al. [ARF+14] developed FlowDroid, a static taint analysis. FlowDroid works as follows: first, it parses the Android manifest, the dex files, and XML files to identify sources, sinks, and entry points. Second, it generates the primary method which it uses to build a call graph for the application. Finally, it runs a taint analysis. Their results show that FlowDroid achieves 93% recall ratio and 86% precision ratio.

3.2 Dynamic approaches

Burguer et al. [BZNT11] present a behavioral-based malware detection system, based on the idea of crowd-sourcing applications. They develop a client, CrowDroid, which users can download from GooglePlay. CrowDroid is responsible for monitoring system calls at kernel-level, and send the data to a centralized server. Users can also submit non-personal, behavioral related data from the applications they use. The remote server parses the data and creates a system call vector for each interaction within each application submitted by a user. They create a dataset for behavioral data for each app. Finally, they apply a clustering algorithm to cluster each dataset. Their analysis showed that *open(), read(), access(), chmod()* and *chown()* are the most used system calls by malware.

Saracino et al. [SSDM18] developed an anomaly detection system named MADAM. MADAM monitor features at four different levels: system calls (kernel level), critical API calls and SMS (application level), user activities (user level), and static application metadata (package level). MADAM estimates an initial security risk by

evaluating the requested permissions and reputation metadata at install time. If MADAM evaluates the application as risky, it inserts it in a suspicious list, which is monitored at runtime. MADAM evaluates these apps against known behavioral patterns in a per-app monitor. Additionally, MADAM preserves a global monitor that collects behavioral data at three levels to learn the current behavior of the device itself. The authors explain the behavioral patterns they use in detail. They collect data from Contagio Minidump, Genome, and VirusShare (2012 - 2015). Their system achieves an accuracy of 96.9% with a false positive ratio of 0.26 at most. However, their solution is not able to detect generic attacks, and botnets can evade it.

3.3 Hybrid approaches

Camacho et al. developed AndroPyTool [MGLCC18] to integrate several popular tools into one single framework. Their resulting tool is a Python framework for extracting features from Android applications in three phases: pre-static, static, and dynamic. The integrated projects in the process include AndroGuard, Droidbox, FlowDroid, VT, and Strace. AndoPyTool was used by the same authors to construct the OmniDroid dataset [MLCC19].

Martinelli et al. [MMS17] present a hybrid tool for detecting malware on Android precisely, named BRIDEMAID. Their tool consists of three steps: static-, metadata- and dynamic analysis. They use static analysis based on n-grams classification, where the frequency of opcodes calculates from the decompiled application. They discard the app if it identifies as malware after this step. Otherwise, the next step executes. The metadata analysis extracts features such as the number of downloads, rating, developer rating, and permissions at install time. Finally, the dynamic analysis derives suspicious activities related to text messages, installed packages, opened connections, and admin authorizations. The dataset they used for experiments consists of 9804 apps from Google Play and 2794 malware samples, belonging to 123 different malware families. The malware samples are from the Genome dataset and the Contagio minidump. They find that their dynamic approach detects malware more accurately than the static method. Overall, they achieve a detection accuracy of 99.7%, which is 2.5% more accurate than the dynamic approach alone and 31% more accurate than the standalone static analysis.

Tong et al. [TY17] suggest a hybrid method where they extract dynamic features and analyze them statically. They obtain individual and sequential system calls from both malware and benign applications, which they use to build patterns of normal and

malicious behavior. For this purpose, they use malware from the Genome project and benign apps from Xi'an JiaoTong University, respectively. They achieve an overall, slightly better on identifying benign apps than malware. However, their approach has two disadvantages. Firstly, their computations execute on a real smart-phone, which is not scalable because of the limited space and capacity. Secondly, they need to update their patterns with newer malware and benign apps regularly.

Arshad et al. [ASW+18] present a 3-level hybrid malware detection model named SAMADroid. Their model consists of static- and dynamic analysis, local and remote hosts, and machine learning. They use a similar approach as in Drebin [ASH+14] for static analysis with a few modifications. As for dynamic analysis, they extract system calls to overcome the limitations of static analysis. Classification of applications utilizes machine learning and executes at a remote server, which then delivers the results to the Android device. To evaluate their solution's performance, they use Drebin's dataset. As stated by the authors, their solution needs newer data to reflect the most recent malware better.

Fratantonio et al. developed Andrubis [LNW+14], an automated hybrid analysis framework. Andrubis consists of three stages: static analysis, dynamic analysis, and auxiliary analysis. The static analysis extracts features from the manifest and the bytecode, while the dynamic analysis monitors the application at the system- and Dalvik level. The auxiliary analysis is used to capture network traffic from outside the operating system and conduct protocol analysis at post-processing. Andrubis was offered to the public as a malware-scanning service (now deprecated). In the period June 12, 2012 – June 12, 2014, the authors received 1,778,997 unique submissions. Andrubis was able to analyze 91.67% of them. As for scalability, Andrubis could analyze 3,500 applications daily. The samples submitted to Andrubis was collected into a dataset and divided into malware and benign. The dataset was further processed to collect statistical information such as file activities performed by malware versus benign apps.

Wong et al. [WL16] propose targeted analysis, which uses static analysis combined with information about the dynamic tool to generate a modest set of inputs that trigger malicious behavior to be detected by the dynamic analyzer. They build a prototype named IntelliDroid[1] that uses a list of targeted API calls specializing in the dynamic analysis tool. This approximation is motivated by the observation that API calls execute the most malicious behavior. The authors demonstrate the possible usage of IntelliDroid by integrating it with TaintDroid [Wil14]. They evaluate it

[1]https://github.com/miwong/IntelliDroid

with malware from Genome and Contagio and show that their solution can detect all privacy leaking malware with no false positives. Compared to FlowDroid [ARF+14], IntelliDroid performs better-considering the detection ratio and false positives. The authors also test IntelliDroid's capability to trigger targeted APIs derived from a hypothetical tool with good results: 70 out of 75 instances were successfully triggered. IntelliDroid is potentially limited by call-graph generation, unrealistic constraints, and malware obfuscation.

3.4 Summary

Table 3.1 presents a summary of the related work.

Table 3.1: Summary of related work.

Ref.	Approach	Limitations	Evaluation
	Static analysis		
[ASH+14]	Deep static analysis. Utilizes machine learning for classification.	Cannot capture new malware easily.	Detection ratio of 93% and 1% false positives.
[FCYS16]	Static analysis. Utilizes machine learning for classification.	High false positive ratio.	Accuracy of 97%, 5% false-positives and 2.7% false-negatives.
[ARF+14]	Taint analysis	Not suitable for real-time analysis.	Recall ratio of 93% and 86% precision ratio.
	Dynamic analysis		
[BZNT11]	Creates behavioral data for each application.	Dependent on network connection. Erroneous when benign apps behave as malware.	Detection ratio 100% on self-written malware. 100% and 85% detection ratio on two real-time malwares.

[SSDM18]	Anomaly detection system that monitors the application at four different levels.	Cannot detect generic malware. Evaded by botnets.	Accuracy of 96.9%, false-positive ratio of 0.26.
Hybrid analysis			
[MGLCC18]	The authors built a hybrid malware analysis tool of existing open-source tools.	No evaluation of the tool. The tool only supports Android version 4.1.	
[MMS17]	Static-, metadata- and dynamic analysis.	Outdated data.	Accuracy of 99.7%. Various benchmark tests.
[TY17]	Build patterns of normal and malicious behavior.	Not scalable. Hard to determine the difference between malware and benign apps. Needs regular updates.	Accuracy of 90%.
[ASW+18]	Static analysis as in [ASH+14]. Collects system calls from dynamic analysis. Uses machine learning for classification of apps.	Needs more recent data.	Measures true positive ratio, false-positive ratio, and accuracy of many machine learning classifiers, in addition to various benchmarking tests.

[LNW+14]	Static analysis, dynamic analysis, and off-device network analysis. Collects data by offering their service as a public scanning service.	Submissions from public sources lack metadata and origin. Their submission system was limited by 8 MB. The dynamic analysis is triggered randomly, while a more intelligent approach could cover more behavior.	
[WL16]	Extracts targeted API calls related to the dynamic tool.	Potentially limited by call-graph generation, unrealistic constraints and malware obfuscation	Compares to FlowDroid.

In our study of related work, we focused on several elements. Firstly, we studied how different authors collected their data. We noticed that most of the authors use pre-generated datasets. Pre-generated datasets are beneficial because they are well-labeled, but many of them are outdated. Another possibility is to collect data similar to Fratantonio et al. [LNW+14], but this approach is significantly time-consuming. Therefore, we decided to construct the dataset ourselves.

Secondly, we focused on system architecture, tools, and approaches. We studied the benefits and drawbacks of different methods and considered tools that we could use further. The hybrid approaches are outperforming static analysis or dynamic analysis alone. In this project, we aim to build a hybrid tool-chain. Thus, we mostly studied hybrid works. Building a pipeline of tools is similar to what Camacho et al. [MGLCC18] did. However, we are also going to use a reputation engine in our pipeline. Unfortunately, we could not find any other work that used a reputation engine in combination with other tools. Nonetheless, both Martinelli et al. [MMS17] and Saracino et al. [SSDM18] have used metadata as a part of their detection system. Metadata is not the same as a full reputation engine, but it has similarities. Thus, we can get inspiration from their architecture when we combine our selected tools.

Finally, we focused on the evaluation of the detection systems. In this case, we studied which evaluation metrics the different authors used. We did this so that we can be sure to include some standard parameters. Using common metrics can ease inter-system comparison. Besides, we studied how well they score so that we can determine if our solution is precise or not.

Chapter 4
Methodology

In this chapter, we explain the overall methodology for answering the research questions. Firstly, we explain how we collect data and how we construct the dataset. We then elaborate on our method for analyzing malware in detail. Evaluating the results is an intrinsic part of the work, and the last section explains different measures for this purpose. Figure 4.1 shows an overview of the methodology.

4.1 Choice of methods

Generally, research is characterized by quantitative- and qualitative methods [Gal08]. While quantitative researchers focus on numbers, descriptive statistics, figures, and illustrations, to show the results of the study, the qualitative researchers deal with descriptions of concepts and perceptions, mainly through interpretations. When we evaluate our system, we will use quantitative measures. It is necessary to include quantitative measures to enable comparisons between diverse malware detection systems. Unfortunately, no universal methodology exists for evaluating malware detection systems to the best of our knowledge. Therefore, we study different evaluation metrics and related work to determine appropriate quantitative malware detection measures for evaluating our system. We select evaluation metrics from the confusion matrix and select relevant resource-related parameters.

To test, verify, and evaluate our pipeline, we need to gather data. Similar to evaluation metrics, data collection has no general procedure or universal requirements. The collected data can have a significant impact on the validity of the pipeline because the threat landscape is under constant change and development. Therefore, we require our data to be recent and general and select an appropriate dataset according to this requirement. We study the literature when we make decisions about the structure of

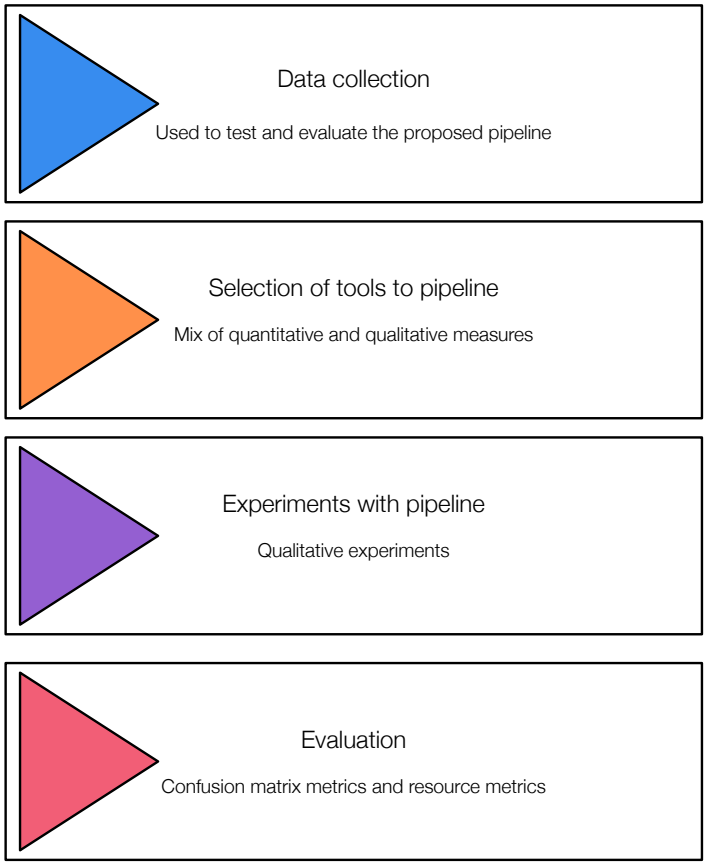


Figure 4.1: Methodology overview.

the dataset.

In the selection of tools, we use a combination of qualitative and quantitative measures. For practical reasons, we *observe* the tool before we test it with quantitative measures. Some of the tools might have limitations that are better captured through observation.

After appropriate tools are selected, they are combined in a pipeline. However, we do not know the best order of these tools. Besides, the output of the tools might be manipulated or combined in diverse ways. Thus, we may find it necessary to conduct several experiments to determine how it affects the performance of our pipeline. We plan these experiments with experimental design. The goal with experimental design is to obtain maximum information with a minimum number of experiments [Jai91]. Planning our experiments is a crucial task, considering the limited time of

this project.

Below follows the overall method that is used to answer each research question:

RQ1: *What should be the trade-off between static- and dynamic analysis to efficiently identify mobile malware?*
To answer this question, we first need to decide our evaluation metrics. These are agreed by studying evaluation metrics used in related work and determining which metrics can capture the overall performance and effectiveness. Then, we measure the metrics by conducting real data experiments.

RQ2: *How can the selected tools be incorporated into a toolchain, and where is reputation-based analysis useful?*
The selection of tools is based on the requirements for the final pipeline. We experiment with different tools to observe if they can fulfill the requirements. We then try different combinations of the pipeline to determine where reputation-based security is useful.

RQ3: *To what extent does the effectiveness of the solution differ on old files compared to recent files?*
We evaluate the pipeline on a live input stream of data. As soon as the data arrives, we analyze it. Then we wait 20 days and re-evaluate it. Finally, we compare the results from the two analyses.

RQ4: *How can the performance of hybrid analysis systems be improved by combining it with results from the reputation-based analysis?*
We answer this question by measuring the evaluation metrics with and without the reputation engine results. We experiment with different combinations to determine if the reputation-based analysis can improve any of our evaluation metrics.

4.2 Dataset

4.2.1 Data collection

High quality, reliable and representative data sets are essential in research activities. In general, there are two ways of collecting malware: setting up a honey-pot or downloading the files from online collections provided by others. A honey-pot is a real or simulated system designed to attract attacks on itself. Because setting up a honey-pot is very time-demanding, the second option is used further.

As Wei et al. [WLR+17] discuss, it is essential that the datasets can meet today's threat landscape. Therefore, the collection of malware samples must meet the following criteria:

- The samples must be collected within at least two years and contain the most recent malware samples. As malware is constantly evolving, the threat landscape is changing over time. To achieve a result that meets today's threat landscape, the malware samples must be as recent as possible.
- The samples must be general, i.e., not belonging to a specific problem area.

Although the AMD dataset[1] is very well-labeled and reliable, we want to include more recent malware samples. However, the authors have published and explained their procedure for collecting this dataset in detail. Thus, we can use a similar method. Since AndroZoo [Li,17] contains a wide variety of applications and continuously updates its database, it is the preferred data set for collecting APKs. Moreover, AndroZoo is not tied to one specific problem area for malware. Hence, it does also provide the opportunity to construct an appropriate dataset for this particular project.

4.2.2 Dataset construction

After accessing AndroZoo, we are given a CSV file containing entries for 10,165,192 mixed samples. Because we require the data to be recent, we begin by extracting all samples in the time interval 2017 - 2020 (as of March 2020). After the initial filtering, we are left with 1,190,420 samples. Most of the applications are unlabeled, meaning that we do not know if they are benign, malware, or from which malware families they belong. However, each file has an entry for "vt_detections" and "vt_date". The value in "vt_detections" means how many antivirus companies that detected the application to be harmful at the time it was submitted ("vt_date"). Unfortunately, these values are mostly outdated or even non-existent for newer files.

A natural next step would be to use VirusTotal reports directly. VirusTotal is a website for scanning hashes, files, or URLs with results from various antivirus companies[2]. For each submitted hash, if it has been scanned earlier by an antivirus tool, we obtain the full VirusTotal report, which includes the first and last time the app was seen, as well as the results from the individual antivirus scans. If the

[1] http://amd.arguslab.org/
[2] https://www.VirusTotal.com/

individual scan was positive, it includes a label, which indicates the malware family. However, the antivirus labels are well-known to be inconsistent [BOA+07][CZR16]. To illustrate this, we select a random hash from our collection. When we submit the hash, seven different antivirus companies detect the application to be potentially unsafe, as shown in Table 4.1. It can be observed that the AV labels are highly incompatible. The remaining 48 antivirus companies did not detect the application to be harmful.

Table 4.1: We selected a random hash and submitted it to VirusTotal. The table shows the different labels that were given by diverse antivirus providers.

Detected by	Label
Cyren	AndroidOS/Trojan.CVIY-3
ESET-NOD32	a variant of Android/Packed.Tencentprotect.B potentially unsafe
Fortinet	Riskware/PackedTencent!Android
Ikarus	Packed.PUA.AndroidOS.Tencent
Jiangmin	AdWare.AndroidOS.gidq
K7GW	Adware (0052b8d61)
SymantecMobileInsight	AdLibrary:Generisk

Re-labelling is a common practice in the academics. Tools and methods exist to gather the various antivirus labels to rename them to a common family name. We have two main options for re-labeling the malware: doing it manually, or by using a tool. Wei et al. [WLR+17] developed a method themselves for doing so. Nonetheless, their approach was very similar to the one used in AVclass [SRKC16]. AVclass is automatic, open-source, and scalable. However, AVclass is limited by the labels it receives as input. In particular, it cannot label samples if at least two antivirus engines do not agree on a non-generic family name. Due to this project's limited time, we use AVclass for labeling and filtering out the non-labeled applications. Another tool for re-labeling is Euphony [Hur17], but it is not as widely used as AVclass. We use Euphonhy as a part of the dataset construction to compare the differences between these two tools.

To proceed, we need to decide the threshold α, which indicates that a sample is a malware if at least α antivirus engines detected it as malware. Unfortunately, there are no rules or standards on deciding this threshold to our awareness. Wei et al. [WLR+17] set this threshold to 50% (28), which gave them a reduction from 1,216,885 applications to 52,520. This means that only 4,32% of the applications passed the filtering. Kakbus et al. [AS18] set the threshold to 20 detections from VirusTotal.

They only lost 7.1% of their collected applications, but they applied other filtering techniques and manual analysis before submitting to VirusTotal. Fratantonio et al. [LNW+14] used a threshold of 5 VirusTotal detections. Others, such as [ASH+14], have selected a subset of antivirus companies based on popularity. They consider an app to be malware when two of their selected antivirus engines agree to say so. This approach was later criticized by [CZR16], which found that selecting a subset of antivirus engines to build a ground truth dataset may lead to more disagreement. Another study [MA14] revealed that multiple antivirus engines are necessary to obtain complete and accurate detections of malware.

Because VirusTotal's Academic API is limited by 20 000 requests each day, we have to rely on the previously fetched detections from AndroZoo. Setting a threshold at all will affect the number of potentially malware samples. The effect for different thresholds is shown in Table 4.2. These values are outdated in some cases, and left undiscovered by some antivirus engines. We observe that 217,872 samples are not yet scanned by any antivirus engine.

Table 4.2: Remaining samples for different values of α.

α	Remaining samples
0	631,883
1	340,667
2	243,548
5	143,062
10	31,788
15	12,301
20	7,191
25	4,003
30	1,709
None	217,871

The detections can be affected by new malware which is not yet recognized by all antivirus engines, or it can be affected by advanced patterns not easily recognized [CZR16]. Figure 4.2 illustrates the number of samples left with an increasing number of VirusTotal detections. We observe that there is no obvious drop that would simplify our decision. Nonetheless, with a higher value of α, we can also have more confidence, although many samples will be lost. Therefore, we chose to set α to 15 detections for our dataset.

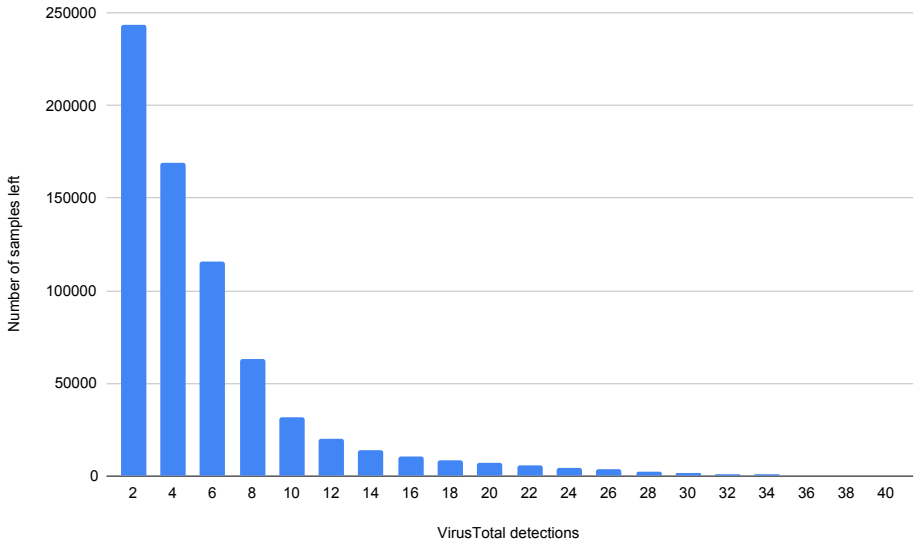


Figure 4.2: The number of samples left with an increasing number of VirusTotal detections.

We further collect benign applications from AndroZoo. We extract applications with zero positive detections. As already stated, some of these detections can be outdated. Therefore, we fetch the most recent reports from VirusTotal to ensure that the applications are benign.

Real-time data

In addition to data collected from AndroZoo, we perform experiments with real-time data. To do so, we receive a live input stream of files recently submitted to Norton LifeLock. To answer **RQ3**, we analyze the files as soon as we receive them, and then wait 20 days and re-analyze them. Then, we compare the results from these two analyses.

4.3 A hybrid malware detection approach

Static solutions are becoming more robust against obfuscated code, but many applications and malware are already using more sophisticated methods for obfuscation [TFA+17]. At the same time, static analysis is effective, scalable, and provides full

code coverage in ways that dynamic analysis cannot. A reputation-based analysis is not affected by code obfuscation and is even more efficient and scalable than static analysis. Nonetheless, reputation-based analysis inherently does not capture any behavior or code coverage. On the other hand, the dynamic analysis does capture an application's behavior, but as with static analysis, malware writers have their techniques to avoid detection. Dynamic analysis is also much more resource-consuming than static- and reputation-based analysis. An overview of the benefits and drawbacks for each method is illustrated in Table 4.3

Table 4.3: Advantages and disadvantages of reputation-based analysis, static analysis and dynamic analysis.

Method	Advantages	Disadvantages
Static analysis	Fast and safe. Can give full code coverage.	Might be evaded by code obfuscation, encryption and packing.
Dynamic analysis	Captures application behaviour. Cannot be escaped by code obfuscation, encryption or packing.	Slow and resource absorbing. Cannot capture the full code. Different evasion techniques exist, such as logic bombs.
Reputation-based analysis	Quick and the least resource consuming method.	Depends on knowledge around or about the app.

A hybrid solution can, therefore, combine reputation-based, static- and dynamic analysis in ways that their added strengths mitigate each other's weaknesses. We set the following requirements for the hybrid pipeline:

- Scalable. The solution must be capable of analyzing hundreds of applications each day.
- Reflect the current threat landscape. The solution must up-to-date and detect the most recent malware.
- Fully automated. Manual analysis in the final solution is not realistic in terms of resources, scale, and time. Hence, we require the solution to be fully automated.
- Precise. The solution must be able to detect malware precisely. Metrics from the confusion matrix is used to evaluate the preciseness of the solution.

We design a solution that consists of a pipeline of reputation-based, static- and dynamic analysis tools. The input to our system is an APK or a folder of APKs. For each analyzed application, the system outputs either benign application or malware.

4.3.1 Selection of tools

Analyzing malware is a time-consuming process, especially when it is conducted manually. Because we require the solution to be automated and scalable, tools must be incorporated in the solution. However, using tools can give a false sense of safety. When using tools, the results are never better than the tool itself. Further, tools can give *false positives* or *false negatives*. That means that a benign app is detected as malware, or that malware was left undetected, respectively. Nonetheless, we have to rely on tools to achieve a realistic solution that can fulfill our requirements.

The selection of tools is based on the requirements for our solution and evaluated with both quantitative measures and qualitative measures. We evaluate the tools with the following metrics:

1. Can the tool be automated, either alone or in combination with other tools? This is a crucial metric for our tools. Otherwise, we cannot fulfill our overall requirements. However, automating the tool may be a time-consuming process. Thus, this metric is tested at last, unless it makes sense otherwise.

2. Is the tool scalable? The scalability of the tools will only be assessed through free-versions of the tool. We measure the scalability in average time per sample when it is possible.

3. Is the tool able to analyze recent samples? Note that we distinguish between *recent* and *new*. In this context, new malware is malware that has never been discovered before and a so-called zero-day malware. We define recent malware as malware that has been seen recently, i.e., in the last year. The objective of this metric is to exclude tools that are outdated or discontinued.

4. Can it give applications a verdict? A verdict in this context is anything that can help us recognize the sample as either malware or benign. This can be a string, a number, a figure, or others.

5. Are there any other limitations, for example, file size limitations, the maximum number of files, or others?

Procedure for finding tools

We start by investigating open-source tools as they are the preferred choice. Open-source tools are the better option in terms of reproducibility and allow us to be aware of the tool's internal workings. Also, we can edit and tweak the code as desired. The open-source tools we evaluate must be maintained projects, or at least updated in the last year. Thus, we ensure that the tool can reflect the current landscape. To find open-source tools, we search on GitHub with a search string built with OR and AND gates. If the tool is a match, we further consider it by reading eventual documentation or other information.

To find scanners and sandboxes, we use various search engines and build a search string. We also read articles and references in the academics, such as in *Android Malware Analysis* [Ken15]. We only assess tools that are free of use. Additionally, we will investigate commercial tools given by Norton LifeLock. The inconvenience with these tools, compared to open-source tools, are the lack of awareness of their internal workings, and that we cannot change or tweak anything.

4.3.2 Planning our experiments

After we have selected appropriate tools, we combine them into a pipeline. We perform experiments with different combinations of the pipeline and measure how it affects the outcome. We aim at performing a minimum number of experiments with maximum obtained information. Firstly, the individual result of each selected tool is studied. When we recognize the weaknesses and strengths of these tools, we can combine them in ways that make sense. All the experiments are qualitative and measured in our selected evaluation metrics.

4.4 Evaluation metrics

We need to determine appropriate evaluation metrics to decide the trade-off between reputation-based-, static- and dynamic analysis. Unfortunately, no universal agreement on evaluation metrics exists, to the best of our knowledge. Therefore, we study related works and the metrics themselves to decide how we can capture our system's performance in the best way. We include metrics related to effectiveness and efficiency. The effectiveness of a system measures the ability to distinguish between malware and benign applications, and efficiency deals with resource allocation, such as memory and CPU [KA15].

4.4.1 Metrics from the confusion matrix

Firstly, we know that our detection system deals with two distinct cases: malware and benign applications. Therefore, our detection system needs to build a Table of all available cases, which is usually called a confusion matrix [HDK17]. The confusion matrix is naturally a 2×2 matrix illustrating whether each instance (malware or benign) has been given a verdict correctly, as shown in Table 4.4.

Table 4.4: Confusion matrix built by a malware detection system.

		Actual class	
		Malware	**Benign**
Predicted class	**Malware**	True positive (TP)	False negative (FN)
	Benign	False positive (FP)	True negative (TN)

True positives (TP): the number of malware samples that are correctly detected.
False positives (FP): the number of benign applications that are erroneously detected as malware.
True negatives (TN): the number of benign applications that are correctly classified.
False negatives (FN): the number of malware samples that are erroneously detected as a benign application.

The most well known evaluation metrics drawn from the confusion matrix are as follows:

Detection ratio

The detection ratio, also named sensitivity or true positive ratio, is one of the main requirements in the design of malware detection solutions [MMF19]. Detection ratio consider malware samples only, and measures how many of them that were actually detected.

$$Detection\ ratio = \frac{\sum TP}{\sum TP + \sum FN} \tag{4.1}$$

Accuracy

Accuracy measures how often the classifier is correct, in total. As with the detection ratio, it is also one of the most common evaluation metrics in malware detection systems.

$$Accuracy\ (ACC) = \frac{\sum TP + \sum TN}{N} \tag{4.2}$$

Precision

Precision considers how many samples that were malware, given that a detection

system verdicts malware.

$$Precision = \frac{\sum TP}{\sum TP + \sum FP} \tag{4.3}$$

False positive ratio
False positives are not directly a risk for the security on the device. However, it can be troublesome if the user is required to interact whenever there is an alarm, and they appear to be false. Hence, a low number of false positives is essential to ensure usability. It is also vital to establish trust between the user and the detection system. If a detection system verdicts known, trusted applications as malware, it is not trustworthy.

The false positive ratio can be measured by using the confusion matrix:

$$False\ positive\ ratio\ (FPR) = \frac{\sum FP}{\sum FP + \sum TN} \tag{4.4}$$

False negative ratio
The false-negative ratio represents how many malware samples that went undetected. Contrary to a false positive ratio, it does pose a security risk to the device.

$$False\ negative\ ratio\ (FPR) = \frac{\sum FN}{\sum FN + \sum TP} \tag{4.5}$$

4.4.2 Resource metrics

Time
Rapid detection and prevention are essential for efficient detection of malware. It affects the complete system performance and is also critical to supply service previously agreed upon (Quality of Service). As can be seen from Table 4.4 this is also the most common resource metric in the related works studied.

Memory, CPU, disk I/O and 2D- and 3D graphics
It is a common practice to benchmark the proposed solution on a real device because malware detection solutions are generally intended for the end-user. Memory, CPU, disk I/O, 2D, and 3D graphics are standard benchmarking tests on Android. Nonetheless, performing these tests requires a full on-device implementation.

Online and offline device
Mobile malware researchers often strive to create a detection system that can run on

a device without an internet connection. But Android devices are still very resource-limited compared to desktop computers, and efficient offline malware detection remains an open research problem.

4.4.3 Evaluation metrics used in related work

Table 4.5: Evaluation metrics from related work divided into metrics from the confusion matrix and resource metrics. The related work was presented in Chapter 3.

Metric	Used by
Confusion matrix	
Accuracy	[FCYS16], [SSDM18], [MMS17], [TY17], [ASW+18]
Detection ratio	[ASH+14], [BZNT11]
False positives	[ASH+14], [FCYS16], [SSDM18], [ASW+18], [ARF+14], [MMS17]
False negatives	[FCYS16], [ASW+18], [ARF+14]
Precision	[ARF+14]
Recall	[ARF+14]
Resource metrics	
Time	[WL16], [ASH+14], [FCYS16], [ARF+14], [LNW+14]
Memory	[SSDM18], [MMS17], [ASW+18]
CPU	[SSDM18], [MMS17], [ASW+18]
Battery consumption	[SSDM18], [MMS17], [ASW+18]
I/O	[SSDM18], [MMS17], [ASW+18]
2D	[SSDM18], [MMS17], [ASW+18]
3D	[SSDM18], [MMS17], [ASW+18]

Table 4.5 gives an overview of evaluation metrics used by related work from Chapter 3. As already stated, there are no general methods for evaluating malware detection systems. To ease inter-system comparison, we attempt to include some of the most common evaluation metrics.

4.4.4 Conclusion

As for performance metrics, we want to include accuracy, detection ratio, and false-positive ratio. Accuracy covers the overall correctness of the system, the detection ratio measures the ability to detect malware, and the false-positive ratio includes the usability of the system. Besides, accuracy and false positives are the most frequently used evaluation metrics in academics. Thus, including these metrics in our project can alleviate the comparison between different systems.

Chapter 5
Experiments and Tools Selection

This chapter explains how we conduct practical work. Firstly, we elaborate on the procedure for data collection. Secondly, we explain the selection of relevant tools that we combine into a final pipeline.

All experiments are conducted on an Ubuntu 18.04 LTS 64-bit as shown in Table 5.1. If any of the tools that we investigate cannot run on an Ubuntu operating system, we use a Windows VM.

Table 5.1: PC specifications.

	Main PC	VM
Operating system	Ubuntu 18.04 LTS 64-bit	Windows 10 64-bit
Memory	31,3 GiB	4048 MB base memory
Processor	Intel® Core™ i7-6700 CPU @ 3.40GHz 8	1 CPU
Disk	503 GB	

Figure 5.1 shows an overview of the scripts we used to conduct the practical work. These are attached in the appendix.

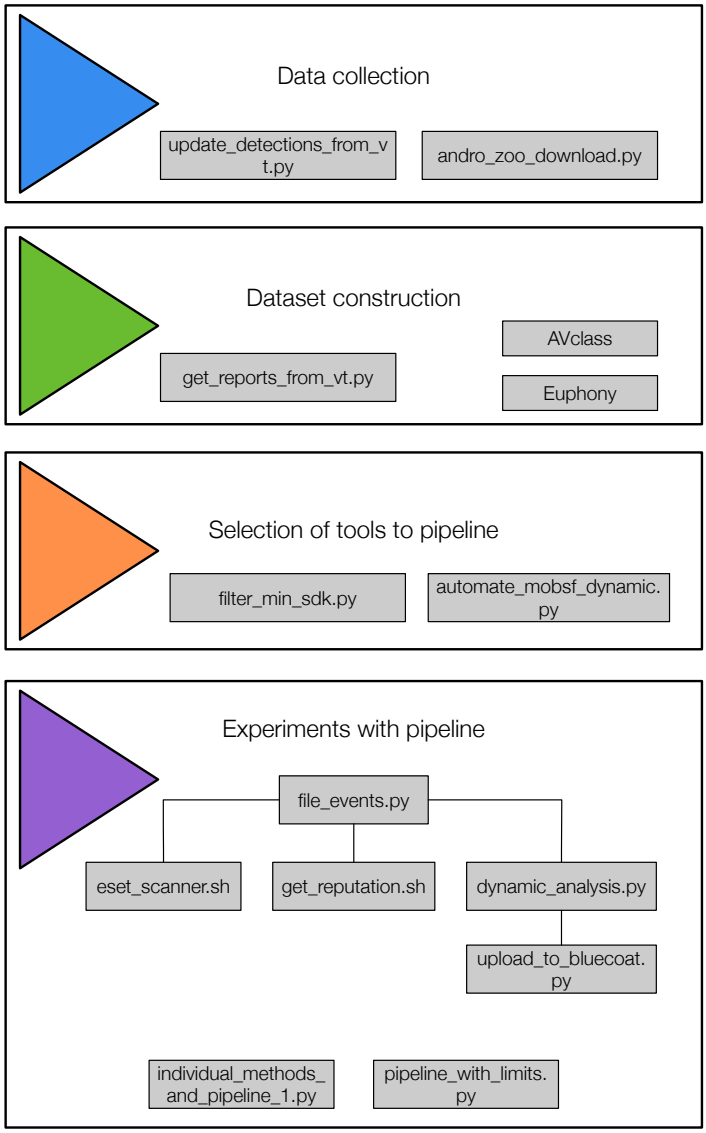


Figure 5.1: An overview of the scripts used to conduct the practical work.

5.1 Data collection

The pre-generated dataset consists of benign applications and malware collected from AndroZoo. We explain the construction of the dataset below. Additionally, we use real-time data collected from Norton LifeLock, which we analyze with the final pipeline.

5.1.1 Dataset construction

We first filter out the applications with 15 or more detections in the time interval 2017 – 2020 from the csv file provided by AndroZoo [Li,17]. After our initial filtering, we were left with 12,301 samples. As previously stated, the VirusTotal detections in the csv file provided by AndroZoo are mostly outdated. Thus, we used Appendix A.3 to update the number of detections and download VirusTotal reports for these samples. This left us with 10,644 samples. To re-label the malware, we used AVclass [SRKC16]. However, a considerable part of the applications was labeled as Potentially Unwanted Applications (PUAs). To ensure diversity, a part of these samples were filtered out with AVClass. Finally, we did some manual analysis of each label to verify its existence and assure that it was a malware family. We did this for each family with more than ten samples. Finally, we were left with the dataset shown in Table 5.2, a total of 2,493 malware samples within 33 different malware families. The samples were downloaded with Appendix A.1.

Table 5.2: Malware families in the pre-generated dataset.

Malware family	Samples	Category
FakeApp	318	Trojan dropper
Piom	295	Trojan
Wroba	235	Banking trojan
SmsSpy	230	Spyware
HiddenAds	228	Adware
Syringe	180	Trojanized Adware
Ramnit	153	Trojan dropper
Triada	126	Backdoor
Shedun	109	Trojanized Adware
Trojandldr	59	Trojan
Virut	45	Virus
Sprovider	44	Adware
FakeInst	42	SMS trojan
Nimda	38	Worm
LockScreen	32	Ransomware
Hqwar	33	Banking trojan
SilentInstaller	32	Riskware
SmsAgent	30	Trojan
Blouns	29	Trojan

Tekwon	24	Trojan
Rootnik	22	Trojan
RemoteCode	21	Trojan
HiddApp	21	Trojan
DroidRooter	19	Riskware
Locker	18	Ransomware
Scar	15	Trojan
Fobus	15	Backdoor
Xiny	14	Trojan
Remco	14	Downloader trojan
Gatf	14	Riskware
Cynos	13	Backdoor
Ztorg	12	SMS trojan
GinMaster	12	Backdoor

We also download benign applications from AndroZoo [Li,17]. These were filtered by extracting applications with zero detections from the csv file from AndroZoo and updated with Appendix A.2. In total, we obtained 2,749 benign applications from various sources, but mainly Google Play. The origins of these applications are as demonstrated in Figure 5.2.

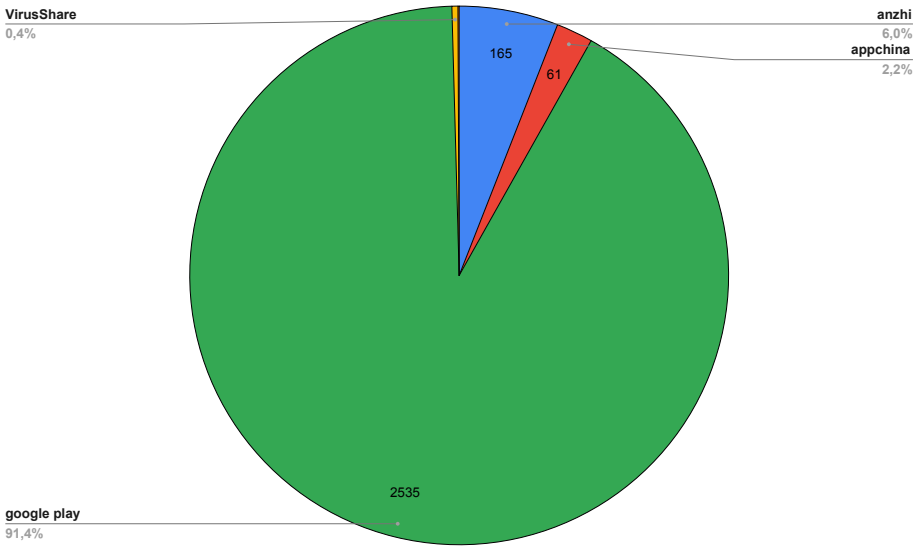


Figure 5.2: Benign applications sources.

5.1.2 Non-filtering of malware samples and challenges with malware labeling

We want to shed light on some of the many challenges with malware labeling. Besides, we want to show that we do not attempt to select samples that our system can easily detect. To do so, we randomly select 2800 samples from our list of sha256s in the time interval 2017 – 2020 with more than 15 detections. Similar to the other dataset, we update the VirusTotal detections with Appendix A.2 and remove apps with less than 15 detections. We do not filter the data anymore. Instead, we show the labels for the data as it is. We use two different tools for this purpose: AVclass [SRKC16] and Euphony [Hur17]. The labels are demonstrated in Table 5.3. We notice that these labels are mostly inconsistent. The most frequent name given by Euphony is "Artemis." Artemis is commonly used by the antivirus company McAfee to name unknown, potential risks [1]. SMSreg is defined as a Potentially Unwanted Application

[1]We read several sample reports and found that McAfee commonly used the name Artemis. We also found it on their support site: https://community.mcafee.com/t5/Malware/What-is-quot-Artemis-quot-trojan-Artemis-502632079A97/td-p/371135

(PUP) by MalwareBytes[2] and Zemana[3], and riskware by f-secure[4] and Sophos[5]. In this case, the label given by AVclass appears to be more consistent. Nonetheless, without a manual inspection, we do not have a clear interpretation of whether it should be, for instance, 206 or 217 samples belonging to "Piom." We include this to demonstrate that the antivirus labels are not reliable and very inconsistent. As a result, using a tool to re-label the malware samples might be erroneous or unreliable. It has also been widely studied and stated in the literature [MA14] [MALM14].

Table 5.3: Top 10 malware families in the non-filtered selected dataset. We show the labels given by both Euphony [Hur17] and AVclass [SRKC16]. We also include the category for each family given by AVclass.

Euphony label	Samples	AVclass label	Samples	Category
Artemis	811	SmsReg	731	Riskware
Dnotua	284	Piom	206	Trojan
Cooee	252	SmsPay	178	Riskware
Piom	217	Dnotua	115	Riskware
FakeApp	78	SmsSpy	86	Spyware
SmsSpy	77	Ramnit	71	Trojan dropper
Triada	73	Triada	70	Backdoor
Ramnit	71	FakeApp	67	Trojan dropper
HiddAd	49	Shedun	59	Trojanized adware
Shedun	45	HiddAd	51	Adware

5.1.3 Real time data

We get access to a live input stream of data from Norton LifeLock. The data is filtered such that only files seen the first time in the last 12 hours are passed to us. We fetch the data from a bucket on Amazon Web Services. When the file arrives, we analyze it with the pipeline and save the results to a file. We fetch this data for three days. Then, we wait 20 days and analyze the same files again.

[2]https://blog.malwarebytes.com/threats/mobile-pup/
[3]https://www.zemana.com/removal-guide/smsreg-malware-removal
[4]https://www.f-secure.com/sw-desc/riskware_android_smsreg_online.shtml
[5]https://www.sophos.com/en-us/threat-center/threat-analyses/adware-and-puas/Android%20Riskware%20SmsReg/detailed-analysis.aspx

5.2 Selection of tools

Initially, we perform experiments on different tools and scanners to evaluate its effectiveness.

5.2.1 Open-source tools

Challenges

Most of the malware detection open-source tools are outdated. For example, CuckooDroid used to be a popular tool for automated malware analysis. It has 475 stars on GitHub and 120 forks. Unfortunately, it was last updated three years ago and currently has 63 reported issues. Newer tools exist. However, most of them are proof of concepts or tied to a specific problem area or data. The initial filtering left us with MobSF, droidefense, and AndroPyTool. These tools are built on other open-source tools to enable automatic malware analysis. MobSF is a popular tool maintained by many active developers. Even though AndroPyTool is not managed anymore, and far less popular, we still chose to test it considering our limited options. Droidefense looks like a promising project but has no official release yet. The master branch was last updated four years ago, and the develop branch has not been updated in 6 months, but we still chose to download the alpha-release of the project.

AndroPyTool

An overview of AndroPyTool is illustrated in Figure 5.3. It consists of three stages: pre-static-, static- and dynamic analysis. The first step extracts features from the application without deep inspection. If we provide a VirusTotal key, the tool can fetch VirusTotal reports and classify the application by using AVClass. The next step includes deeper static analysis by using the Python Framework AndroGuard[6] for selecting various static features, and FlowDroid [ARF+14] to run taint analysis to follow the information flow. The last step is the dynamic analysis, where AndroPyTool uses DroidBox[7] and Strace[8]. AndroPyTool can be downloaded from GitHub/AndroPyTool. It is easy to install and deployed through a Docker container. This is beneficial because it can run on any operating system.

Pre-static analysis. The pre-static analysis outputs the file hash, the number of detections from the downloaded VirusTotal report, and the AVclass label. Unfortunately, the labeling does not work as intended, since it outputs "android" for almost

[6]https://github.com/androguard/androguard
[7]https://github.com/pjlantz/droidbox
[8]https://linux.die.net/man/1/strace

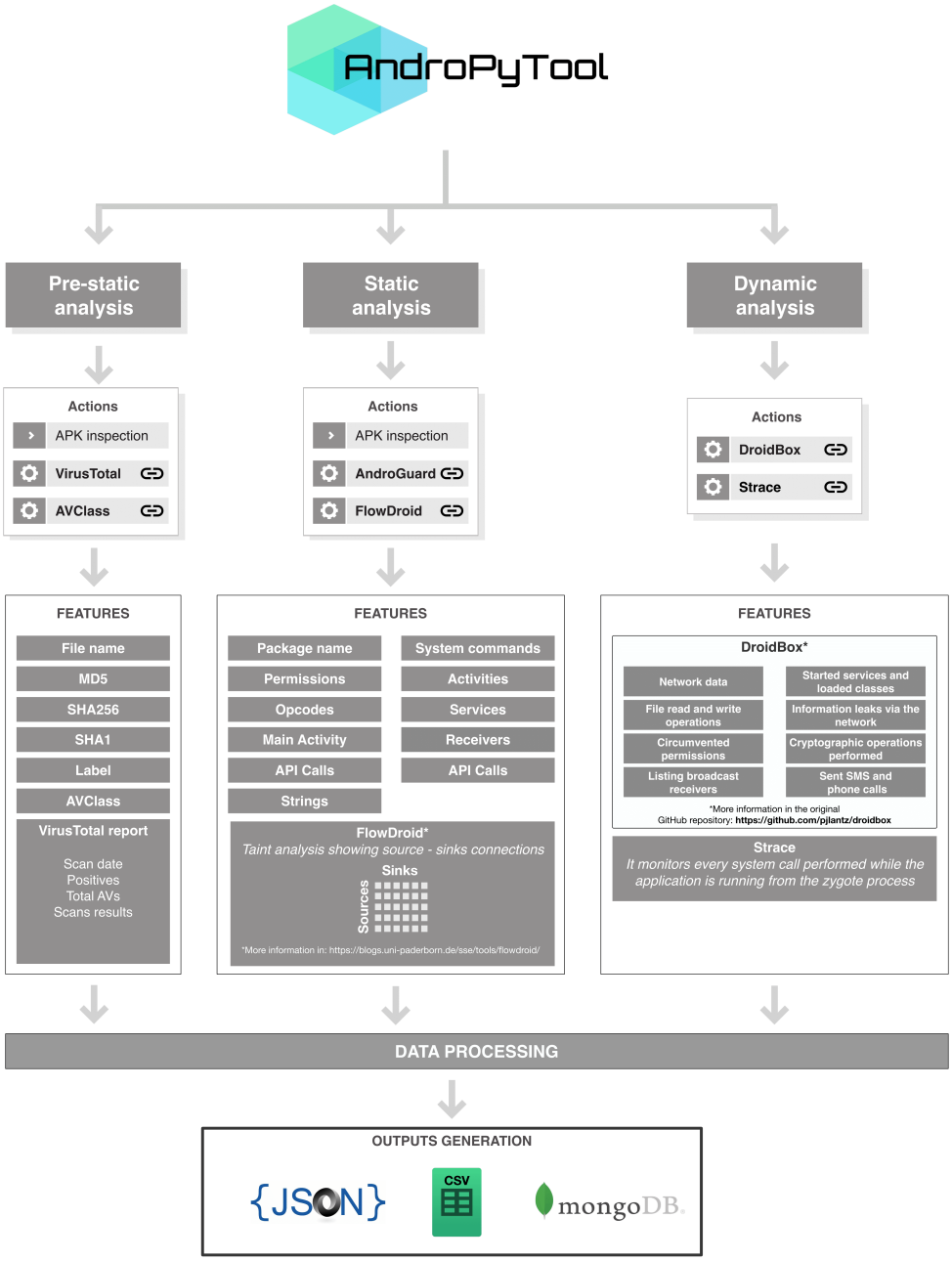


Figure 5.3: An overview of AndroPyTool [MGLCC18].

all files. Two reasons can cause this problem: either there is a bug in AndroPyTool, or AVclass is not able to recognize the labels from VirusTotal. Nonetheless, since we used AVclass to label the dataset, we conclude that the bug is in AndroPyTool.

Static analysis. The static analysis collects class name, opcodes, declared permissions, API calls, strings, API packages, system commands, intents, activities, services, and receivers. However, a drawback with AndroPyTool is the time it uses to analyze applications statically. FlowDroid runs taint analysis, which is not applicable for real-time analysis. FlowDroid alone spent almost 43 hours analyzing 1498 apps, or 1.715 minutes per application.

Dynamic analysis. As mentioned, DroidBox is the underlying tool for dynamic analysis in AndroPyTool. DroidBox runs applications in an Android emulator to monitor events such as file access, network traffic, SMS activities, cryptography usage, started services, and dynamically loaded dex files. It uses Monkeytool[9] to generate pseudo-random streams of user events such as clicks, touches, or gestures, and several system-level events. Unfortunately, Droidbox was built for Android 4.1. This level corresponds to SDK version 16. The minimum SDK is decided by the developer and express an application's compatibility with one or more versions of the Android platform, utilizing an API Level integer. Despite its name, minimum SDK is used to specify the API level and not the SDK version [SDK19]. As AndroPyTool does not automatically filter out applications, we filter out the apps with a SDK lower than 16. This corresponds to removing 978 applications out of 5467. We use AndroGuard to conduct the filtering, see appendix B.1. Figure 5.4 shows the minimum SDK versions for the dataset.

Another drawback with Droidbox is the fixed bugs. Currently, it has 24 reported issues in Github, and it is no longer maintained. We experience several of these bugs while testing it, which causes it to crash unexpectedly. The crashes make it less applicable for large-scale automated analysis of applications.

[9] https://developer.android.com/studio/test/monkey

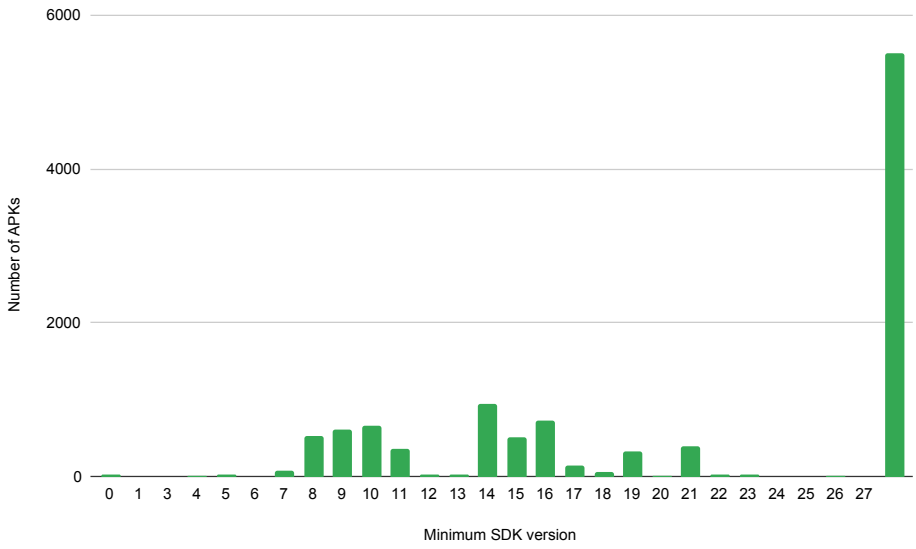


Figure 5.4: Minimum SDK versions for the dataset.

MobSF

MobSF is an all-in-one cross-platform tool for security assessment and malware analysis, which downloads from GitHub/MobSF. It is somewhat more work to install MobSF compared to AndroPyTool. For static analysis, it is possible to run it in a Docker container, but not for dynamic analysis. Therefore, we install it from the source. MobSF uses Genymotion[10] as a virtual runtime environment for dynamic analysis, which enables higher Android versions compared to AndroPyTool. Currently, MobSF supports Android 9. Figure 5.5 demonstrates an overview of MobSF.

Static analysis. MobSF uses jadx to decompile the dex files of the application java files and AndroGuard to extract various features. As with AndroPyTool, MobSF fetches detections from the VirusTotal reports if the key is present. Nonetheless, it does not provide labeling. A local web interface can be accessed where the user can read reports and upload files. Access through API is also possible, which enables mass static analysis.

[10]https://www.genymotion.com/

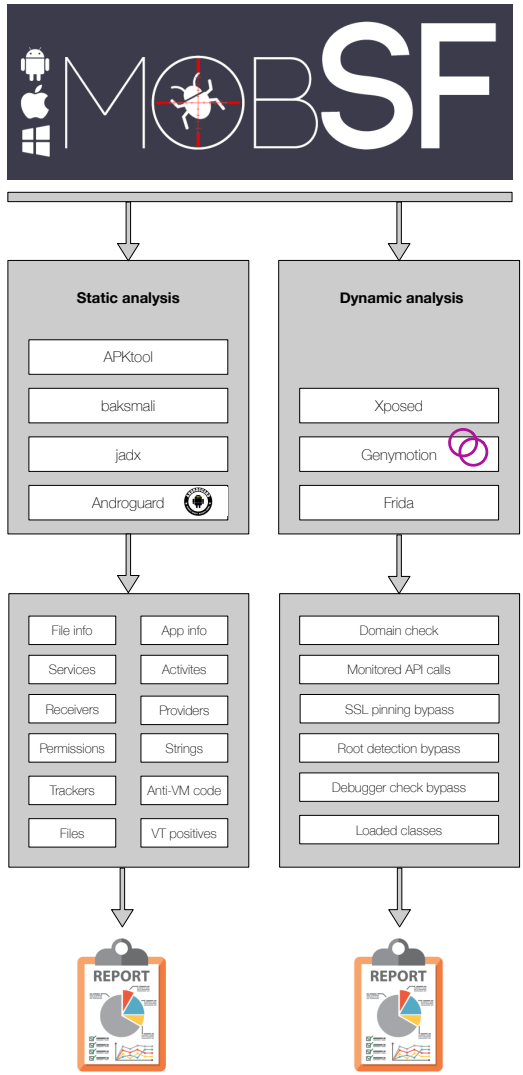


Figure 5.5: MobSF architecture.

MobSF already provides a script for mass static analysis. The script automatically uploads all the applications in a provided folder and then scans them. The report for each scan shows overall security scores, VirusTotal detections, average Common Vulnerability Scoring System score, tracker detection, and details about what it found. Although the tool provides details specific to malware detection, such as VirusTotal detections, it is more specified on finding security vulnerabilities. It does not classify the application as benign or malware automatically.

Dynamic analysis. The dynamic analysis provided by MobSF is alleged to be user-assisted, and not automated by default. Therefore, we write a script to automate it, see Appendix B.2. With this automation, the dynamic analysis performs a malware domain check for each URL used by the app and monitors API calls. The dynamic analysis does not fetch features such as system calls, installed packages, or suspicious activities. System calls, for instance, is a commonly used feature for dynamic malware analysis, such as in [BZNT11], [TY17], [ASW+18], [SSDM18]. Overall, the dynamic analysis alone does not provide us with enough details to decide whether the application is a malware or not.

Droidefense

We download droidefense from its GitHub repository. It provides scripts for building and compiling, but these result in errors. We try to edit the project settings, but the project is dependent on old modules, and some do not exist anymore. Thus, we do not test the tool further.

5.2.2 Scanners

In this section, we assess different free scanners against our evaluation criteria. The scanners must be possible to run on a desktop, either through a web interface or in a desktop application. Therefore, we did not test scanners available in Google PlayStore.

Online scanners

AVCUndroid[11] is a fast, online, static scanner. It accepts a maximum of 7 MB files, which would exclude almost all the data samples. Therefore, we do not test it further. The same reasoning applies to Dr. Web Online[12], which only allows for 10 MB files. Further, the online static analyzer VirScan[13] accepts a maximum of 20 MB files and

[11] https://undroid.av-comparatives.org/
[12] https://vms.drweb.com/online/
[13] https://www.virscan.org/

is particularly slow. Finally, Kaspersky[14] offers static analysis online. It is fast and limited by a file size of 50 MB, adequate for all the collected samples. Unfortunately, mass-scale assessments are not achievable with the scanner, as the analysis cannot be automated.

Desktop scanners

We are given a license to Norton LifeLock's commercial desktop scanner, Norton Security[15]. Norton Security is a powerful scanner with no strict limitations in terms of filesize, processing time, or others. It runs on OSX and Windows. Initially, we try it on OSX, but the application does not provide any possibility for command line integration after the latest OSX update (Catalina). Command-line integration is necessary for automation and fetching the results. Therefore, we try it further on Windows, as the scanner does provide a few options for execution through the command line on Windows. However, even if it is executed from the command prompt, it still renders a graphical interface where user interaction is needed. To overcome this issue, we try to use AutoIt[16], a powerful scripting language to automate Windows applications. To finish the scan and export the results, we need to click "export" in the graphical interface. Unfortunately, the buttons are invisible, which means that they cannot be clicked. We try other tools for automation as well, which gives the same result. Another possibility is to export the full security history from the command line. The results from the security history provide fewer details, but enough to make a decision. However, for the most recent scan to be included in the security history, the graphical interface still requires user interaction.

We also try Eset's desktop scanner[17], which is free within the first 30 days. Eset runs on any operating system, including Ubuntu. Similarly to Norton's scanner, it does not have any strict limitations. Also, Eset offers extensive command-line integration, which enables full automation. It keeps a signature database locally and performs signature-based detection. It regularly updates the signature database whenever it has an internet connection. Although this method is efficient against known malware, it cannot recognize new malware and may be circumvented by obfuscated code. We tested the tool with and without an internet connection to determine if the results were different. We expected that it might fetch pre-calculated results from dynamic analysis from a database, but the results were the same. Concerning execution time,

[14] https://www.kaspersky.no/
[15] https://no.norton.com/norton-security-antivirus
[16] https://www.autoitscript.com/site/
[17] https://www.eset.com/

the scanner is fast enough. It uses 3.55 seconds per application on average, which is way faster than, for instance, Flowdroid's 1.75 minutes per application.

Indifference from AndroPyTool and MobSF, all the scanners were automatically giving samples a verdict, so the samples were differentiated as either malicious or normal. Another difference from the open-source tools is that we are not aware of these scanners' underlying technology. Hence, giving an explanation of the inner workings or fixing bugs is not possible.

5.2.3 Sandboxes

A sandbox provides a safe and isolated environment to execute suspicious code and is thus the preferred way of conducting a dynamic analysis.

AMAaaS

AMAaaS[18] is an online sandbox that provides both static- and dynamic analysis. It allows for larger files than AVCUndroid and Dr. Web Online, with a limitation of 20 MB. Still, this would exclude many files. Additionally, AMAaaS deploys a queue system for analyzing files, which is very slow, and therefore not adequate for efficient analysis.

Joe sandbox

Joe Sandbox[19] is another sandbox that provides both static- and dynamic analysis. It does not suffer from a strict file size limitation and enables up to 100 MB files, which is enough for all the collected samples. Unfortunately, the free subscription is limited to ten submissions each day, and it does not allow us to use their API or chose either static- or dynamic analysis. To answer the research questions, we need to be able to distinguish the different analysis methods. Because of the limited time of this project, we also have to analyze more than ten samples each day.

Norton LifeLock's sandbox

We get access to Norton LifeLock's sandbox. The sandbox performs dynamic analysis, and reputation-based analysis on the URLs the application attempts to visit. Figure 5.6 demonstrates an overview of the architecture. Note that the internals are trade secret and that we only extracted the available information. The dynamic analysis is based on DroidBox, but with custom modifications. To determine if an

[18] https://amaaas.com/
[19] https://www.joesecurity.org/

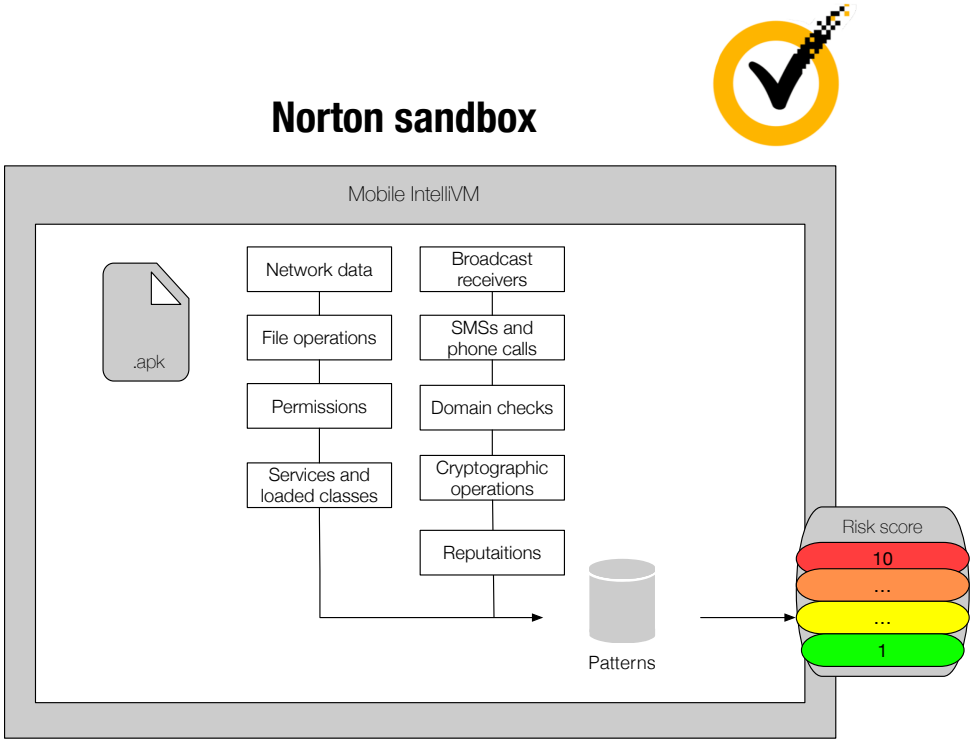


Figure 5.6: An overview of Norton LifeLock's sandbox. The internals are trade secret, and the figure only demonstrates the information we were able to extract.

application is malware, it attempts to match the behavior of the application against pre-determined patterns. If no patterns match, it returns a security score of one. If patterns belonging to the most severe group are found, it returns ten. Applications are considered as malware if the score is above seven. For example, if the application connects to sites that are known for advertisement, the application will receive a security score of five. If the application contains a high priority SMS receiver, it will obtain a security score of eight.

We access the sandbox through a remote API. It easily allows for automatic, mass-scanning of files. For each file, we upload the APK, create a task, and fetch the results once it finishes. The only limitation is the processing capacity. Unfortunately, we cannot decide how many apps that are analyzed simultaneously. For each app, we have to wait at least two minutes. We can only run a few of them in parallel. However, we do expect the dynamic analysis to be more time consuming than static

analysis.

5.2.4 Reputation-based analysis

We assess Norton LifeLock's commercial reputation engine. It is accessed through an API and fetches stored reputations for the samples we submit. Only one request is necessary for each sample, and the response is fast ($<$ 1 second). The reputation engine returns a score between -120 and 120. According to Norton LifeLock, score mapping is as follows:

- Scores above 110 are defined as "high good."
- Scores above 30 are "medium good."
- Scores from 10 to 30 are "low good."
- Scores between 0 and 10 are "neutral."
- Scores from -30 to 0 are "low good."
- Scores lower than -30, but higher than -110 are "medium bad."
- Scores lower than -110 are "high bad."

The output score is based on different aspects that either contribute to a high or low reputation. For example, applications within the gaming category with poor performance, that leaks information through SMSs, obtains a low score. On the other side, a productivity app with no leaks and excellent performance will receive a high score.

We did not assess more reputation-based analysis tools, as there were no other tools available for free. However, the reputation engine from Norton LifeLock fulfills all of our requirements. It is scalable, automatable, outputs a score for each application, and can analyze the files we give it.

5.2.5 Conclusion

We tested several open-source tools, scanners, sandboxes, and commercial tools, and evaluated its effectiveness against metrics explained in Chapter 4. A summary of these tools and their capabilities are given in Table 5.4. None of the tested tools fulfills these requirements perfectly. AndroPyTool is an automatic tool for malware detection, but it cannot dynamically analyze applications with a higher API level

than 16 because of its underlying technology. It is also slow and erroneous. MobSF can dynamically analyze newer versions of Android and can be automated. Another benefit of MobSF is that it is still being maintained, in contrast to the other open-source tools within automatic malware analysis. However, it does not provide us with enough relevant information to decide whether the app is malware. Instead, it is a more suitable tool for penetration testing.

With regards to sandboxes, Norton LifeLock's sandbox is the best option. The other sandboxes we tested are too limited. Norton LifeLock does fulfill the main requirements but does not score very well on scalability caused by its long execution time. Nonetheless, we still include it in our pipeline.

Scanners are the best option concerning verdicts. However, the difficulty with these scanners is the strict limitations, either in terms of file size, automation, or little flexibility. In general, the online-scanners are better suited for a one-time scan rather than a mass-scale analysis. The desktop scanners are far less restricted. They still suffer from little flexibility and the fact that we do not know the underlying technology. Thus, we cannot tweak or modify any code. Neither can we perform debugging to solve potential problems. The most suitable scanner was Eset's scanner. As it is the tool that is closest to fulfill the main requirements, it will be used further for static analysis.

We include Norton LifeLock's reputation engine in addition to Esets's scanner and Norton LifeLock's sandbox. We did not have many options for reputation-based analysis, but Norton's reputation engine does fulfill our requirements.

Table 5.4: Summary of the tested tools and their capabilities. The selected tools are highlighted.

Tool	Automat-able	Verdict	Scalable	Recent samples	Limitations
Open-source tools					
AndroPyTool	✓	✗	✗	✗	Dynamic analysis limited by API level 16.
MobSF	✓	✗	✓	✓	More suitable for penetration testing.
Scanners					
AVCUndroid	–	✓	✗	✓	7 MB filesize.

Dr. Web Online	–	✓	✗	✓	10 MB filesize.
Kaspersky	✗	✓	✗	✓	
Norton Security	✗	✓	✓	✓	
Eset	✓	✓	✓	✓	Cannot detect new malware, or obfuscated malware.
VirScan	–	✓	✗	✓	20 MB filesize. Slow.
Sandboxes					
Norton's sandbox	✓	✓	✓	✓	Cannot paralelize more than three instances.
AMAaaS	–	✓	✗	✓	20 MB filesize. Slow.
Joe Sandbox	–	✓	✗	✓	Can't distinguish static- and dynamic analysis.
Reputation engines					
Norton's reputation engine	✓	✓	✓	✓	

5.3 Final pipeline

The final pipeline consists of Norton's reputation engine, Eset's scanner, and Norton's sandbox. We are receiving a live input stream of data, and thus, we write a python script that can listen to new files in a folder. The data samples were directly downloaded to that folder. The script contains an observer, which observes the changes, and an event handler that handles the file event. We use the Python library

Watchdog[20] for this purpose. Once the event handler is called, it uses the tool AndroGuard[21] to compute the sha256 of the file. The sha256 is needed to request results from the reputation engine. Then, it runs the reputation-based analysis, static analysis, and dynamic analysis with our selected tools. Because the processing of data is a time-consuming process, we use three threads to process the files. It would be faster to parallelize the tasks even more, but we are limited by our allowed processing capacity on the sandbox. Finally, the results are written to a CSV file holding the scores for each method, the sha256 and filename, and the execution times. For our pre-generated dataset, we write the results for the benign applications to one file, and similarly one file for the malware. We use the result files to experiment with different combinations and save the combined results in new files, together with a summary. The exact combinations are decided after we have studied the individual performance of the tools, which is explained and illustrated in Section 6.1.1. The scripts for the full pipeline can be seen in Appendix C.

[20] https://pythonhosted.org/watchdog/
[21] https://github.com/androguard/androguard

Chapter 6
Results and Discussion

This chapter presents the results of the experiments. We begin by presenting the results from our experiments on the generated dataset and discuss our observations. Then, we show results from the non-filtered dataset and compare the results. Finally, we introduce and discuss results derived from real-time data experiments. We also include limitations of our pipeline and discuss the requirements as defined in Chapter 4.

6.1 Filtered dataset

Table 6.1 show the calculated evaluation metrics for each analysis method alone. We have included the analysis method by itself to determine individual performance.

Table 6.1: Results for each analysis method alone. The results are from experiments with the pre-generated filtered dataset.

Method	Accuracy	Detection ratio	False positive ratio	Time (s)
Reputation-based analysis only	0.9588	0.9146	1.092×10^{-3}	0.8478
Static analysis only	0.8954	0.7806	3.639×10^{-4}	4.333
Dynamic analysis only	0.6300	0.2314	8.370×10^{-3}	330.0

As can be observed, the reputation-based analysis is the method with the lowest execution time. It simply fetches the result from a remote database in one request. On the other side, it is always dependent on an internet connection, reducing scalability.

Further, the reputation-based analysis has very few false-positives, but the accuracy is not impeccable, which is mainly caused by its detection ability. A possible reason for the lower detection ratio is that the reputation engine does not have enough reputation data on some files. Therefore, it will give the file a neutral score. As we have set the limit for malware to below zero (neutral), this will result in some undetected files. Figure 6.1 demonstrates the scores returned from the reputation-based analysis. We observe that 1859 samples are in the neutral category. For the 2284 apps with a score lower than zero, only three of them are false alarms. All samples belonging to the malware families "Nimda" and "DroidRooter" were left undetected by the reputation engine.

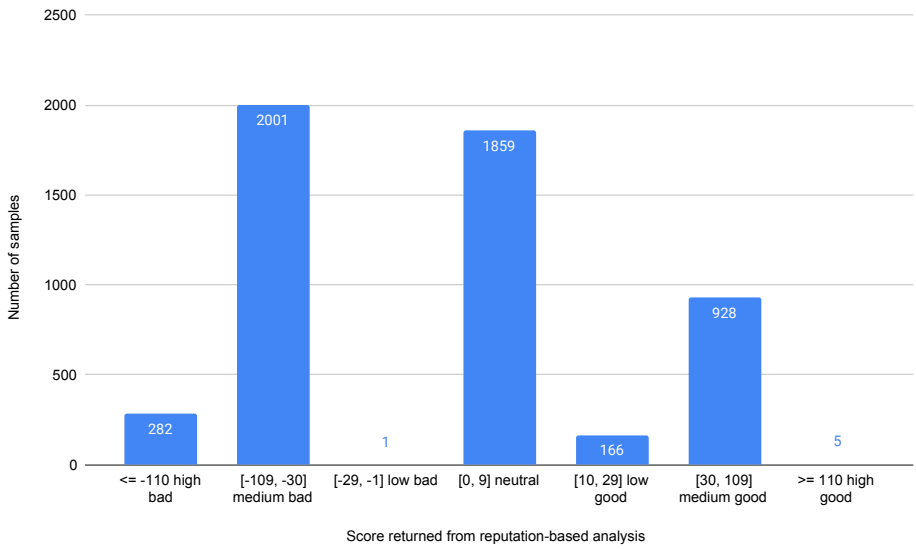


Figure 6.1: Reputation scores for the filtered pre-generated data set.

Its signature-based approach causes the lower detection ratio for the static analyzer. A signature-based method performs very well for known malware but fails to detect unknown malware. It can also be affected by obfuscated code. Besides, the approach results in almost zero false-positives (one false positive occurred for the entire data set). When a sample's signature matches the signature of known malware, we can almost be entirely sure that it is malware. The malware families that were poorly detected by the signature approach were "Piom," "Syringe," "DroidRooter," and

"LockScreen." Further, the execution time of the static analyzer is dependent on a single database search. The signature database exists locally, and when the static analyzer receives a data sample, it merely searches for that signature in its database. Hence, the average execution time for benign applications is higher than for malware because it has to search through all entries in the database.

Indifference from the reputation-based- and static analysis, the dynamic analysis has a very high execution time. We expect an extensive execution time because the dynamic analysis is a more resource-intensive task. Besides, it is necessary to run the application for a while to capture its behavior. Others, like Lalande et al., found that they had to use 312 seconds on average for each sample to conduct dynamic analysis [LTLG18]. Unfortunately, the dynamic analyzer does not perform any better than the reputation-based engine or the signature-based approach. We expect the dynamic analyzer to have some strict rules, causing the low detection ratio. When rules are defined, it is always a cumbersome task to find the balance between too rigid and too slack rules. Immensely strict rules will result in a lower detection ratio, while overly loose rules will give many false positives. In this case, the dynamic analyzer's strategy may be to keep the false positives to a minimum, i.e., by setting strict rules. Another cause of the meager detection ratio is that some malware may apply evasion strategies to prevent getting caught by the dynamic analyzer. However, because the detection ratio is extremely low, another explanation may be that the sandbox crashes. We extracted the families that it cannot detect to understand why the dynamic analysis results in a poor detection ratio. We derive that *none* of the samples belonging to "FakeApp," "GinMaster," "Gatf," "Scar," and "Remco" is identfied as malware by dynamic analysis. Very few of the samples in the family "Piom" (5 of 295), HiddAd (7 of 228), SmsSpy (44 of 230), Ramnit (3 of 150), SilentInstaller (3 of 30) is detected by dynamic analysis. The only family the dynamic analyzer can identify with good results is "Wroba," where it caught 234 samples out of 235. The poorly detected malware families did not give us any clear indication about the meager detection ratio. Thus, it strengthens our suspicion about system crashes.

6.1.1 Combining the tools

The dynamic analysis is by far the most resource-consuming approach and consumes significantly more time than the other methods. It is also the method with the lowest detection ratio by far. However, in a few cases, it may be a better method for detecting malware. Therefore, we place it at the end of the pipeline in all cases.

Table 6.2 shows results from different combinations of the pipeline. We include results for two out of three analysis methods and one where all of the approaches are included, named P#1. The architecture is inspired by Martinelli et al. [MMS17]. P#1 is defined as follows: If the reputation engine outputs a security score of less than 0, we assume that the app is malware and does not process it further. If the score is above 0, the static analyzer further process the application. Provided that the output from the static analyzer is not malware, it is finally processed by the dynamic analyzer. Otherwise, the file is categorized as malware. For an overview, see Figure 6.2.

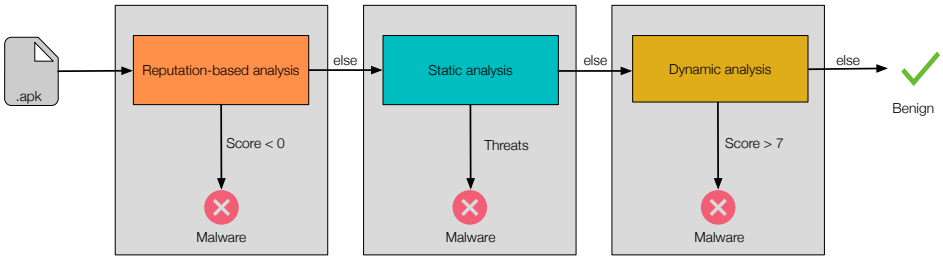


Figure 6.2: P#1: Pipeline architecture.

As for the results where only two analysis methods are included, it works similar to P#1 but with one less step. Note that we could have placed the static analyzer before the reputation-based analyzer, but the result would be the same. Because the reputation-based analyzer is faster than the static analyzer, it is first in order when it does not affect the results.

Table 6.2: Results for different combinations of the analysis methods when the malware samples are filtered.

Method	Accuracy	Detection ratio	False positive ratio	Time (s)
Reputation-based- and static analysis	0.9834	0.9663	1.092×10^{-3}	3.801
Reputation-based- and dynamic analysis	0.9655	0.9374	9.0976×10^{-3}	226.2
Static- and dynamic analysis	0.9158	0.8327	8.734×10^{-3}	242.0
P#1	0.9861	0.9807	9.098×10^{-3}	221.7

P#1 performs slightly better than reputation-based- and static analysis only. The dynamic analyzer is able to detect 36 samples, mainly from the "Syringe" family, that the other tools cannot detect. Thus, detection ratio is so far the best. With P#1, only the samples shown in Table 6.3 are left undetected. On the other hand, the false positive ratio is increased, and the average execution time is also particularly higher than with reputation-based- and static analysis. This is mainly because samples that are benign have to undergo all the analysis methods.

Table 6.3: Samples left undetected by P#1. P#1 is the option with the highest detection ratio.

Family	Number of samples left undetected
DroidRooter	19
SilentInstaller	11
LockScreen	10
RemoteCode	4
Xiny	2
Triada	1
Remco	1

Further, we observe that the pipeline is mostly erroneous when analyzing samples from the families "DroidRooter," "SilentInstaller," and "LockScreen." The DroidRooter family are applications that can perform binary exploits to gain root access. Although it is classified as an exploit, it can also include apps that users deliberately download to achieve root-access of their device. Therefore, we would have to perform manual analysis to determine whether it has malicious intent or not. The LockScreen family is a category of ransomware that can lock the user's screen and display some personal information collected from the device, and then ask the user to pay. We submitted a few of the undetected samples of LockScreen to VirusTotal and found that there was significant disagreement about the naming of these samples. Still, several of them contain suspicious requests, permissions, and bundled executables. Although the naming might not be correctly identified, these should still have been detected, at least as potentially unwanted applications (PUPs). As for the last family, "SilentInstaller," the naming is generic. Nevertheless, a common observation of all the samples belonging to that family, is that they contain one or more executables hidden in the code.

6.1.2 Limit sensitivity

With P#1, benign applications have to undergo all the analysis steps, causing an excessive execution time. Although this approach results in a high detection ratio, we experiment with different combinations to reduce the number of false positives and to reduce the execution time, while maintaining a high detection ratio. Expressly, we have experimented with different limits for the score returned from the reputation-based analysis. In P#1, we only used a limit for malware. We now present results where we also set a constraint for benign applications. In this case, we also tried combinations where the static analyzer is placed first.

The reputation engine returns an integer between -220 and 220. Thus, if we were going to test every possible combination, it would require 57,840 experiments. However, we can apply the same principals as explained in [Jai91]. Instead of having each integer as a score level, we chose to reduce them to 22 levels, where each level is an interval of ten. Further, from Figure 6.1, we observe that very few samples obtain a score between -1 and -30. We also know that the files with a reputation score lower than zero mostly are malware, with a reasonable false positive ratio. The most considerable uncertainty in the verdict from the reputation engine derives from the samples with a score above zero. Therefore, we set the limit for malware to a constant < 0. We are then left with 22 experiments. Although this is a considerable reduction, we believe that it will not affect the results significantly.

Figure 6.3 presents the accuracy, detection ratio and $1-$ false positive ratio for the different limits. The number of false positives is the same for both orders, but the number of false negatives is lower when we place the static analyzer first. This observation holds until the limit reaches 110, where the results are the same. In both cases, the best accuracy and detection ratio is reached when the limit is set to 110. Then we achieve the same result as with P#1. The flaw with P#1 and high limits, is the increasing number of false positives. Besides, the time increases when we increase the limit, which is illustrated in Figure 6.4. The differences between the orders are not significant. The increased run-time is because more samples need to be analyzed by the dynamic analyzer.

We have also included a table with the results for the different limits in Appendix D, Table D.1.

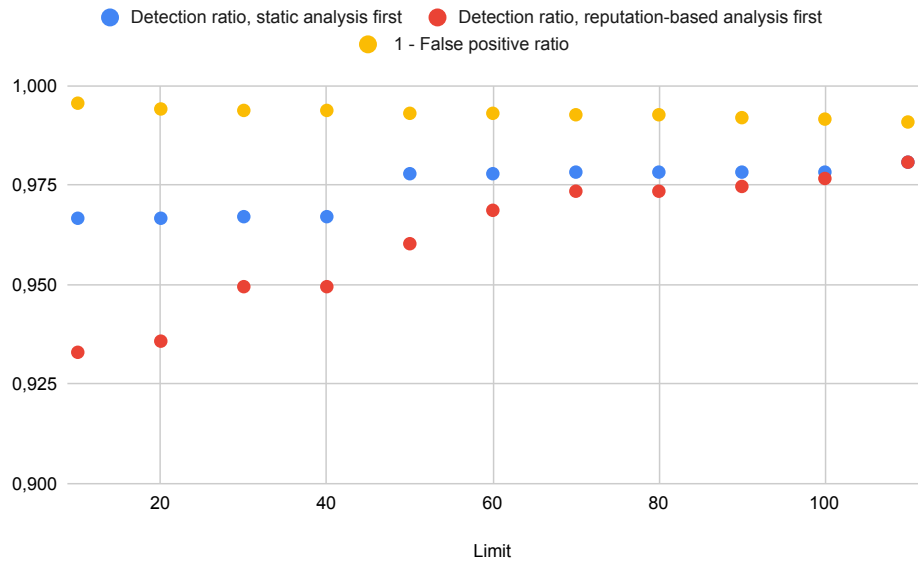


Figure 6.3: Accuracy, detection ratio and 1− false positive ratio for different limits.

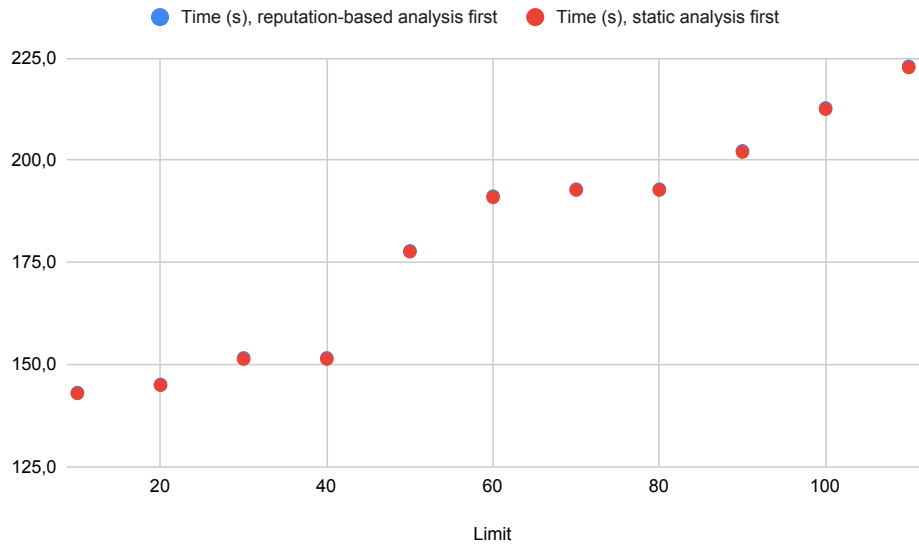


Figure 6.4: Average time per sample for the different limits.

6.2 Non-filtered samples

Table 6.4 show the calculated evaluation metrics for each analysis method alone when the malware samples are selected without any filtering.

Table 6.4: Results for each analysis method alone when the malware samples are not filtered.

Method	Accuracy	Detection ratio	False positive ratio	Time (s)
Reputation-based analysis only	0.8760	0.7805	1.092×10^{-3}	0.8745
Static analysis only	0.8156	0.6314	3.636×10^{-4}	4.333
Dynamic analysis only	0.8371	0.6809	8.370×10^{-3}	292.0

Compared to the filtered dataset, the dynamic analysis seems to perform much better on this dataset. Nonetheless, we note that it detects almost all samples from the "SmsReg" family (668 out of 731). It does not recognize the other families any better than in the previous dataset. This scrutiny demonstrates why it is crucial to balance datasets.

Both the reputation engine and the static analysis perform worse on this dataset than the filtered one. The reputation engine fails to detect 436 out of 731 samples in the "SmsReg" family, while the static analysis cannot recognize 512 out of 731 samples from the same family. The static analyzer is also particularly bad at detecting the "Piom" family, where it missed 193 out of 206 samples. However, the non-filtered dataset contains a considerable fraction of families in the "gray zone," including riskware and Potentially Unwanted Applications (PUAs). For example, the SmsReg family is in the riskware category, which constitutes 27% of the malware samples. Gray zone applications might be harder to detect because there is a more delicate line between malware and non-malware. As already stated, a manual inspection would be necessary to confirm the correct verdict of these applications.

Table 6.5 demonstrates the same combinations that we included in the previous section. Note that the malware samples are the difference and not the benign samples, which remains the same. Thus, the false-positive ratios are identical to previously. Results from experiments with the filtered dataset implied that the dynamic analysis might be skipped, considering the meager detection ratio and excessive time-consumption.

Table 6.5: Results for different combinations of the analysis methods when the malware samples are not filtered.

Method	Accuracy	Detection ratio	False positive ratio	Time (s)
Reputation-based- and static analysis	0.8915	0.7830	1.092×10^{-3}	3.584
Reputation-based- and dynamic analysis	0.9925	0.9941	9.098×10^{-3}	186.2
Static- and dynamic analysis	0.9484	0.9051	8.734×10^{-3}	206.0
P#1	0.9929	0.9949	9.098×10^{-3}	181.7

The results from this dataset imply otherwise; the reputation engine and the scanner are not efficient in detecting gray zone applications, the "SMSreg" family in particular. Although the reputation engine and the scanner perform worse than previously, we observe that the full pipeline, P#1, achieves a higher accuracy on this dataset than the filtered dataset. This is mainly because the sandbox was able to detect samples that the other tools could not recognize.

We do not consider these results as valid because of the considerable imbalance in the dataset. However, it demonstrates the potential of such an approach. If we could find a dynamic analysis that can perform reasonably well on all datasets, the pipeline would be able to detect samples with very high accuracy.

6.3 Real time data

We now present results from experiments on real-time data. The goal of these experiments was to determine how our pipeline performs on recent files compared to older files. To achieve our goal, we initially analyzed samples seen for the first time in the last 12 hours by Norton LifeLock. The same data were re-analyzed 20 days later. This section presents and discusses differences in results from the two analyses.

6.3.1 VirusTotal detections and malware families

Figure 6.5 demonstrates the number of VirusTotal detections derived from the real-time data for the first- and the second analysis. We observe that on the second

analysis, more anti-virus engines detected the files that already were detected. We further use the VirusTotal detections when we discuss the results.

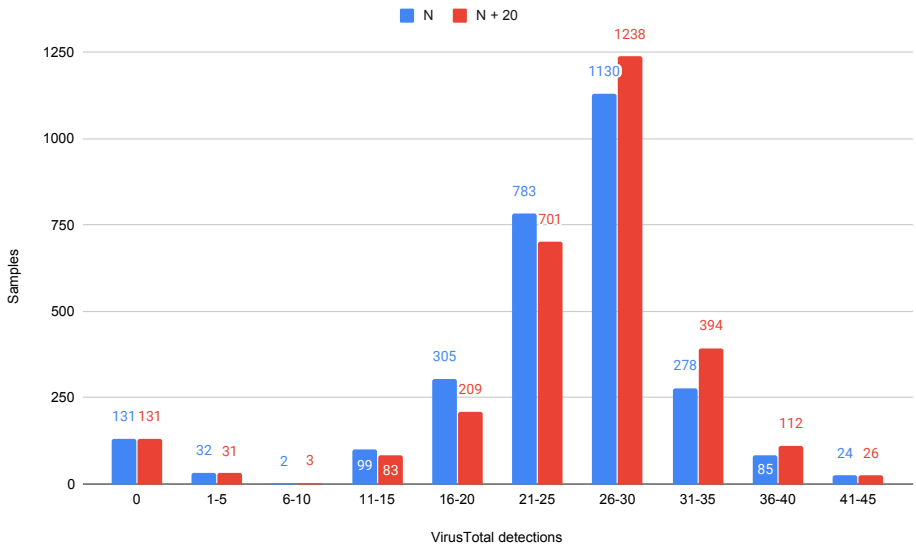


Figure 6.5: VirusTotal detections for the real-time dataset, including both the first result (N) and the second result (N + 20) for the same dataset. It is 20 days between the two results.

Table 6.6 presents the 15 most frequent malware families extracted from the real-time data with AVclass [SRKC16]. We have also included the original year of development of each malware family. We observe that most of the malware from the real-time data are belonging to older families than for the pre-generated dataset. In fact, the samples in the pre-generated dataset seems to match the description of the current threat landscape we gave in Section 2.4.3 better than the real-time data. Unfortunately, we were not able to understand why this is the case.

Table 6.6: Top 15 malware families derived from the real-time data. The families were derived during the secondary analysis.

Malware family	Samples	Category	Year Developed
FakeInst	353	SMS Trojan	2010 [Ken15]
Shedun	309	Trojanized adware	2015[1]
DroidKungFu	266	Trojan	2011 [Ken15]
Opfake	166	SMS trojan	2011[2]
BaseBridge	153	SMS trojan	2011 [Ken15]
Boxer	122	SMS trojan	2012 [Ken15]
GingerMaster	95	Trojan	2011 [Ken15]
Plankton	80	Backdoor	2011 [Ken15]
HiddenAds	63	Adware	2014 [FMCB16]
Kmin	55	Trojan	2011[3]
DroidDreamLight	46	Backdoor	2011 [Ken15]
Batterydoctor	41	Trojan	2011 [Ken15]
Iconosys	39	SMS trojan	2012[4]
FakeAdBlocker	39	Adware [Che20]	–
Lotoor	34	Risk tool[5]	–

6.3.2 Pipeline results

Table 6.7 shows the results from the pipeline, divided into VirusTotal detections. Overall, the tool-chain performed similarly in the two analyses. The applications that had zero VirusTotal detections from the first analysis also had zero VirusTotal detections from the second analysis. However, one of the applications that had zero VirusTotal detections was judged as malware by the pipeline. The verdict was given from dynamic analysis in both cases. We conducted some manual analysis on that particular application to understand why. Then, we observed that it matched a malicious pattern defined by the dynamic analyzer related to SMS receivers. It does not necessarily mean that the application was malicious, but rather, it demonstrates the drawback of this approach. Defining perfect patterns of malicious behavior is

[1] https://en.wikipedia.org/wiki/Shedun
[2] https://www.f-secure.com/v-descs/trojan_android_opfake.shtml
[3] https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Trojan:AndroidOS/Kmin.A
[4] https://www.juniper.net/security/auto/includes/mobile_signature_descriptions.html
[5] https://threats.kaspersky.com/en/threat/Exploit.AndroidOS.Lotoor/

Table 6.7: Results from the pipeline for the first and second analyses. We show the output from the pipeline, considering the number of VirusTotal detections.

VirusTotal detections	Pipeline results, first analysis	Pipeline results, second analysis
0	131 benign, 1 malware	131 benign, 1 malware
1–14	23 benign, 63 malware	25 benign, 54 malware
15 or more	3 benign, 2649 malware	5 benign, 2654 malware

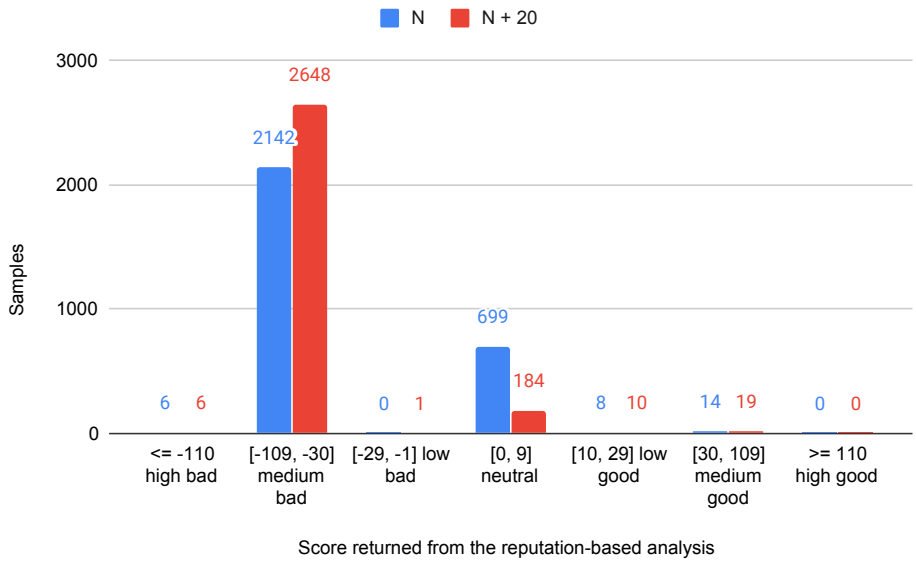


Figure 6.6: Reputation scores for the real-time data set. The graph includes results from the first analysis (N) and the second analysis (N + 2) of the same dataset.

impossible. With a pattern-matching approach, a benign application that includes behavior similar to malware can be falsely judged as malware.

Further, we also did some manual analysis on the applications that went undetected (benign) by our pipeline but had more than 15 VirusTotal detections. The files that were undetected by our pipeline in the first analysis were also left undetected in the secondary analysis. Besides, two more files were undetected in the secondary analysis. The three files that went undetected in both cases were identified as "riskware." Recall that riskware are legitimate apps that *can* be exploited by a person with malicious intent [Rey20]. They are not designed with malicious intent but contain weaknesses

that can be exploited. In Section 2.4.1, we defined malware as any application with malicious intent. However, we also stated that applications that are in the riskware category should be considered by its context. Thus, filtering out riskware may be correct, following our definition. As for the two new undetected files from the secondary analysis, one of them was recognized as riskware. The other was identified as a variant of Droidsheep. Droidsheep is the equivalent of the desktop hacking tool Firesheep [Ken15]. It is capable of hijacking social networking sessions on services such as Facebook and Twitter [Gol12]. Thus, the pipeline produced one more false negative in our secondary analysis.

The results from the reputation-based analysis alone are shown in Figure 6.6. We notice that fewer applications belong to the neutral category in the second analysis compared to the first analysis. This demonstrates that reputation-based analysis functions better when it has more knowledge about a sample.

6.4 Requirements discussion

Scalability. Our implemented solution is not scalable. The time spent in the queue for dynamic analysis would increase significantly with a growing number of users. It is necessary to drastically increment the number of simultaneous tasks performed at the dynamic analyzer to improve the scalability of our system. We could also store the results from the dynamic analyzer for a specific time, such that only new applications and applications with updates would go through this step. Another aspect is the dependency on an internet connection. Many potential users may not have an internet connection at all times or a fragile connection. We have not tried to implement the whole system at an Android device. However, the device's capacity would probably be reduced if we were going to apply it on a mobile device. Device capacity is a common problem with anti-virus on mobile phones, as they are far more restricted than a desktop computer [TFA+17].

Reflect the current threat landscape. We have included recent malware in our pre-generated dataset and a live input stream of apps first seen in the last 12 hours. Our results show that the tool-chain can detect recent malware with reasonably good results. However, from Figure 6.6, we saw that the reputation-based analysis performed better when it had more knowledge about the applications. We also know that the signature-based approach used in the static analysis only can recognize known malware. Our sandbox defines patterns of malicious behavior, which also requires updates to keep track of changes in the threat landscape. Therefore, we do

not fulfill this requirement perfectly. To improve, we could have included a machine learning approach.

Fully automated. Our solution is fully automatic, and therefore fulfills our requirement on automation.

Precise. We have measured the preciseness of our solution in metrics from the confusion matrix, specifically accuracy, detection ratio, and false-positive ratio. We achieve the best accuracy and detection ratio with P#1, or with P#2 when the limit is > 110. Then, we accomplish an accuracy of 98.61%, which is not perfect, but reasonably good. However, the lowest false positive ratio was obtained with static analysis alone. Nonetheless, the overall results with the static analyzer alone are not acceptable. We have to consider the preciseness from the overall results, where all of our chosen metrics are examined. With P#1, or with P#2 when the limit is > 110, the false positive ratio is still beneath 1%. Note that we achieved a higher detection ratio and thus accuracy when we tested the pipeline on the randomly selected samples. However, we do not consider this an utterly valid result because 27% of the entire malware dataset belongs to one family.

6.5 Limitations

Unbalanced dataset. As addressed in [RDG+12], malware families should be balanced over the dataset. The most frequent malware family in the filtered dataset contains 318 samples, while the least common malware family includes only 11 samples. Nonetheless, the dataset was left unbalanced not to loose too many samples.

Evasion. We used Eset's scanner for static analysis. It is based on a signature-based approach, which is efficient for known malware, but not capable of detecting new malware and can be evaded by obfuscated code. Further, dynamic analysis is based on pattern matching. Thus, it can be evaded by malware that behaves differently than already defined. Also, malware writers can apply different evasion techniques against it. For example, malware can detect that it is being run within a sandbox environment.

Reproducability. Because we included commercial tools in the pipeline, experiments with the same tools are challenging to reproduce. Nevertheless, we have explained our procedures for collecting data, selecting tools, and performing experiments.

Dynamic analysis. The selected sandbox performs notably worse than the other

tools that we selected. We can only try to estimate why it gives such a low detection ratio because we do not have full insight into the tool. Some of the false negatives are probably caused by sandbox evasion and too strict pattern matching. However, because the detection ratio is significantly low, we believe that DroidBox is also a part of the problem. We evaluated DroidBox in Section 5.2.1 and found that it causes the system to crash in a substantial amount of cases. Besides, it supports a maximum of Android version 4.1. Norton's sandbox is based on DroidBox, which might be a reason for the low detection ratio. The meager detection ratio was not recognized when we tested it because we did not calculate the evaluation metrics for each tool. Nonetheless, we were left with very few options for dynamic analysis. Therefore, we believe that there is a great need for a new open-source tool for dynamic analysis of malware on Android.

Reliability. We used VirusTotal as an oracle for determining if a sample is a malware or not. Lalande et al. [LTLG18] found that this is not entirely reliable, especially for recent data. The anti-virus detections we rely on tends to use machine learning or signature-based approaches. Although machine learning provides scalable mechanisms for detecting malware, its success relies on accurate training data [Bac15]. The best option is to conduct manual analysis as a part of the work during dataset construction, but this is not feasible with thousands of samples. Our study found several apps that we could not positively determine if it was malware or not without manually inspecting it. Therefore, we realized that manual examination should have been a part of the work when we constructed our dataset. Zhou et al. [ZJ12] did a manual inspection of the malware when they formed the Malware Genome project. The Genome dataset is one of the most trusted and used datasets that exist, but it is now deprecated. Wei et al. [WLR+17] performed a manual analysis of samples within each variety after their automated filtering. Although this is not utterly reliable, it is a good alternative. We leave a manual review as a future task.

6.6 Comparison with related work

Compared to BRIDEMAID [MMS17], their method resulted in much higher accuracy for the dynamic analysis alone (0.9720), than ours. However, their malware was gathered from different, older sources (Genome [ZJ12] and Contagio), and they had a much higher fraction of benign apps than us. With a significant part of benign apps, each analysis method would perform much better, considering that they are more erroneous when detecting malware. Therefore, the comparison is not entirely fair. This also demonstrates the need for more universal evaluation methods. The

authors of BRIDEMAID [MMS17] also found that the dynamic analysis was the most accurate analysis method. We found that the reputation-based analysis outperforms both static- and dynamic analysis on our data set. Nonetheless, the dynamic analysis that we used may have caused the system to crash.

The best option for comparing with related work would be to use tools from similar work directly. Specifically, we could try other authors' implementation with our dataset and environment. This would ensure the same surroundings for both projects and allow for an impartial comparison. Wong et al. did this when they evaluated their tool, IntelliDroid [WL16]. The difficulty with such an approach is that the implementations often are owned by a company, or one can face technical issues [ARF+14]. Thus, it can be surprisingly time-consuming. Unluckily, we did not have additional time to perform experiments with other implementations in this project.

Chapter 7
Conclusion and Future Work

7.1 Conclusion

Malware is an ever-evolving threat to security on Android devices. Research on Android malware is not new, and various solutions and products to detect and protect from malware exist. However, there is a constant need to build solutions that can keep track of the changing threat landscape. Android is the most popular mobile operating system, making it the favorable target for malware writers. Compared to its most significant opponent, iOS, Android is also far more open and less restricted. Besides, third-party market stores make malware distribution easier.

During this project, we built a hybrid tool-chain for detecting malware on Android. Hybrid solutions appear to be the more prominent approach, which was why we built a hybrid pipeline. We selected tools from several criteria, and we evaluated open-source tools, free scanners and sandboxes, and commercial tools. The final tool-chain included Norton LifeLock's reputation engine and sandbox, and Esets's scanner. Including a reputation engine in a hybrid approach has not gained attention from the academics before, at least not to our knowledge.

We performed experiments with both real-time data and pre-generated data. The pre-generated data was collected from recent years to ensure that our pipeline was up-to-date. We collected malware samples and divided them into two different datasets. For one of the datasets, we filtered the samples so that we could ensure diversity. For the other dataset, we collected 2800 applications, updated the VirusTotal detections, and analyzed them. Without filtering, we saw that the dataset included a family that constituted 27% of the dataset. The pipeline performed better on the non-filtered dataset than the filtered dataset. The increased performance was mainly due to the sandbox, which could better analyze the largest family of the non-filtered dataset.

Nonetheless, we did not consider the results from the non-filtered dataset as valid.

Further, we also received real-time data from Norton LifeLock. The data were filtered such that files first seen in the last 12 hours were delivered to us. We analyzed the data as soon as it arrived. Then, we waited for 20 days and re-analyzed it. Overall, the pipeline results from the two analyses were similar. The most considerable difference was from the reputation engine, which got more confidence in the secondary analysis of the files. Compared to the first analysis, fewer files were in the neutral category in the second analysis.

We evaluated the tool-chain with chosen metrics from the confusion matrix, explicitly false positive ratio, accuracy, and detection ratio. Besides, we included time as a resource metric. We observed that with an increasing detecting ratio, the number of false positives and the run-time also increased. For the pre-generated filtered dataset, our best effort in terms of detectability gave a detection ratio of 98.08% and an accuracy of 98.61%. In that case, the false-positive ratio was still below 1%, and the average time per sample was 221 seconds.

The project encompassed several challenges. One of the most significant challenges was to collect and construct the dataset. For example, deciding the threshold α, which indicates that a sample is a malware if at least α antivirus engines detected it as malware, was a hard task. We had no clear understanding of which value α should have been. Besides, labeling the malware was also a cumbersome task, as the antivirus labels tend to be highly incompatible. In the end, we realized that the way we collected and constructed the data was not utterly reliable.

7.1.1 Answer to research questions

RQ1: *What should be the trade-off between reputation-based, static- and dynamic analysis to identify mobile malware efficiently?*

The exact trade-off depends on the agreement with potential users. If run-time is crucial to the user, then we should choose a pipeline with static analysis first and limit $>= 10$. With these conditions, a more significant trade-off is given to static analysis and reputation-based analysis, which are the faster methods.

On the other side, our best effort in terms of detectability resulted in a detection ratio of 98.08% for the pre-generated filtered dataset. If security is more critical to the user, this is the preferred approach. In this case, 93.2% of the malware samples were detected during the reputation-based analysis, 5.2% during static analysis, and

1.5% during dynamic analysis, which is by far the most time-consuming approach. For the real-time data, all the malware samples were detected during static- and reputation-based analysis. However, in a few cases, the dynamic analysis should receive a more significant trade-off. For example, the dynamic analyzer detected the "SmsReg" family much better than the other tools. Nonetheless, this approach is fast and accurate for a higher fraction of malware and slower for benign applications. When an application is harmless, it has to go through all the analysis methods.

RQ2: *How can the selected tools be incorporated into a tool-chain, and where is reputation-based security useful?*

After we selected our tools, we were left with Norton LifeLock's reputation engine and sandbox, and Eset's scanner.

Experiments with the filtered dataset show that reputation-based analysis, together with only the static analysis, performs almost as good as any solution with all the methods. The difference in accuracy is only 0.27%. However, the dynamic analysis scored particularly low on the detection ratio. We suspect that it caused the system to crash for many of the samples. Therefore, we do not believe that this is a general observation. For the non-filtered dataset, the findings were otherwise. Then, the dynamic analysis was able to detect more samples, and we found reputation-based analysis and dynamic analysis to be the more useful methods.

In any case, we found the reputation-based analysis to be useful. It increased the detection ratio and accuracy, produced very few false-positives, and is incredibly fast. Therefore, the tools should be incorporated with reputation-based analysis and static analysis first, and dynamic analysis at the end. The dynamic analysis is the most resource-consuming approach, and it is therefore conducted at last, only when the other methods cannot detect samples.

RQ3: *To what extent does the effectiveness of the pipeline differ on old files compared to recent files?*

We received a live input stream of data from Norton LifeLock, which we analyzed directly. Then, we waited for 20 days and re-assessed the same files. Overall, our results show that the pipeline does not differ significantly in these two analyses. We recognized one sample as a false positive for both analyses, which was the same file in both cases. Also, we got one more false negative in the second analysis than the first, but also one less false positive. However, most of the samples from the real-time data belong to known malware families, as shown in Table 6.6. Thus, we expect that

the tools recognize the malicious code and behavior from previous samples.

RQ4: *How can the performance of hybrid detection systems be improved by combining it with results from reputation-based security?*

Experiments with the static- and dynamic analysis only resulted in an accuracy of 0.9158, a detection ratio of 0.8327, and a false-positive ratio of 8.734×10^{-3}. When we combine these with results from the reputation-based analysis, the accuracy is increased by 7.03%, and the detection ratio is increased by 14.8%. Additionally, the run-time is decreased with more than 20 seconds on average. On the other side, the false positive ratio is also slightly increased. Thus, we conclude that the performance of a hybrid detection system can be improved in terms of accuracy, detection ratio, and decreased run-time. However, we cannot guarantee that our results generally hold, as is may differ for other tools and approaches. Again, we state that the dynamic analysis performed much worse than expected, which may cause the results to be distinct for other systems.

7.2 Future work

Most importantly, we believe that there is a considerable need for updated, open-source malware analysis tools. In particular, a framework for automated malware analysis is needed. Making such a tool is an immense task that would require enormous effort. Such frameworks do exist but are either deprecated or not maintained in the last years. As Android is continually pushing new features and APIs, and the threat landscape is changing, these tools are too outdated. The existing frameworks are mostly based on other open-source tools. Within dynamic analysis, there has not been a new extensive tool for years. Some tools can be used in static analysis to simplify the process, such as AndroGuard. Therefore, we believe that a new open-source tool within dynamic analysis is the prioritized task in future work.

There is also a need for a universal evaluation method. It is a cumbersome task to compare systems when inter-systems use various evaluation metrics and gathers data with distinct approaches. Thus, we hope someone can take on the difficult but essential task to define a comprehensive evaluation method.

As previously stated, we also believe that it is necessary to generate a new, reliable dataset for researchers. We did create a dataset in this project, but it was realized that manual inspection of the malware samples should have been a part of the work. Thus, we leave this as a future task.

References

[Abr16] Ajin Abraham. Appsec eu 2016: Automated mobile application security assessment with mobsf. Rome, Metropolitian City of Rome, Italy, 7 2016. OWASP Foundation.

[ARF+14] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM SIGPLAN Notices*, 49, 06 2014.

[AS18] Kursat Aktas and Sevil Sen. Updroid: Updated android malware and its familial classification. In *Nordic Conference on Secure IT Systems*, pages 352–368. Springer, 2018.

[ASH+14] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, and Konrad Rieck. Drebin: Effective and explainable detection of android malware in your pocket. 02 2014.

[ASKA16] Saba Arshad, Munam Ali Shah, Abid Khan, and Mansoor Ahmed. Android malware detection & protection: A survey. *International Journal of Advanced Computer Science and Applications*, 7(2), 2016.

[ASW+18] S. Arshad, M. A. Shah, A. Wahid, A. Mehmood, H. Song, and H. Yu. Samadroid: A novel 3-level hybrid malware detection model for android operating system. *IEEE Access*, 6:4321–4339, 2018.

[Bac15] Rekha Bachwani. Better malware ground truth: Techniques for weighting anti-virus vendor labels. 10 2015.

[BOA+07] Michael Bailey, Jon Oberheide, Jon Andersen, Z. Morley Mao, Farnam Jahanian, and Jose Nazario. Automated classification and analysis of internet malware. In Christopher Kruegel, Richard Lippmann, and Andrew Clark, editors, *Recent Advances in Intrusion Detection*, pages 178–197, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[BZNT11] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: Behavior-based malware detection system for android. pages 15–26, 10 2011.

[Car19] Bluetooth-worm:symbos/cabir. https://www.f-secure.com/v-descs/cabir.shtml, 2019. Last accessed: 2020-01-20.

[Che20] Victor Chebyshev. Mobile malware evolution 2019. https://securelist.com/mobile-malware-evolution-2019/96280/, 02 2020. Last accessed: 2020-06-04.

[CJS+05] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *2005 IEEE Symposium on Security and Privacy (S P'05)*, pages 32–46, May 2005.

[Cle19a] J. Clement. Average number of new android app releases per day from 3rd quarter 2016 to 1st quarter 2018. https://www.statista.com/statistics/276703/android-app-releases-worldwide/, 03 2019. Last accessed: 2020-04-21.

[Cle19b] J. Clement. Development of new android malware worldwide from june 2016 to may 2019. https://www.statista.com/statistics/680705/global-android-malware-volume/, 09 2019. Last accessed: 2020-04-01.

[CZR16] J. Caballero, U. Zurutuza, and R.J. Rodríguez. *Detection of Intrusions and Malware, and Vulnerability Assessment: 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings.* Lecture Notes in Computer Science. Springer International Publishing, 2016.

[Dat19] Digital 2019: Global digital overview. 01 2019. Last accessed: 2020-02-22.

[DMAS16] H. M. Deylami, R. C. Muniyandi, I. T. Ardekani, and A. Sarrafzadeh. Taxonomy of malware detection techniques: A systematic literature review. In *2016 14th Annual Conference on Privacy, Security and Trust (PST)*, pages 629–636, Dec 2016.

[Ele14] Nikolay Elenkov. *Android Security Internals.* No Starch Press, Inc, 245 8th Street, San Francisco, CA 94103 USA, 1 edition, 10 2014.

[FCYS16] H. Fereidooni, M. Conti, D. Yao, and A. Sperduti. Anastasia: Android malware detection using static analysis of applications. In *2016 8th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–5, Nov 2016.

[FMCB16] Hossein Fereidooni, Veelasha Moonsamy, Mauro Conti, and Lejla Batina. Efficient classification of android malware in the wild using robust static features. 2016.

[Fre84] Fred Cohen. Experiments with computer viruses. 1984.

[Gal08] Galyani Moghaddam, Golnessa and Moballeghi. How do we measure use of scientific journals? a note on research methodologies. 2008.

[GO18] Nana Kwame Gyamfi and Ebenezer Owusu Owusu. Survey of mobile malware analysis, detection techniques and tool. pages 1101–1107, 11 2018.

[Gol12] Steve Gold. Wireless cracking: there's an app for that. *Network Security*, 2012(5):10 – 14, 2012.

[HAL20] Legacy hals. https://source.android.com/devices/architecture/hal, 1 2020. Last accessed: 2020-03-07.

[HDK17] Tarfa Hamed, Rozita Dara, and Stefan C. Kremer. Chapter 6 - intrusion detection in contemporary environments. In John R. Vacca, editor, *Computer and Information Security Handbook (Third Edition)*, pages 109 – 130. Morgan Kaufmann, Boston, third edition edition, 2017.

[Hur17] Hurier, Médéric and Suarez-Tangil, Guillermo and Dash, Santanu Kumar and Bissyandé, Tegawendé F and Traon, Yves Le and Klein, Jacques and Cavallaro, Lorenzo. Euphony: harmonious unification of cacophonous anti-virus vendor labels for android malware. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 425–435. IEEE Press, 2017.

[Jai91] Raj Jain. *The art of computer systems performance analysis - techniques for experimental design, measurement, simulation, and modeling.* Wiley professional computing. Wiley, 1991.

[KA15] Dr. Gulshan Kumar Ahuja. Evaluation metrics for intrusion detection systems-a study. *International Journal of Computer Science and Mobile Applications*, 11, 06 2015.

[Kar03] M. Karresand. Separating trojan horses, viruses, and worms - a proposed taxonomy of software weapons. In *IEEE Systems, Man and Cybernetics SocietyInformation Assurance Workshop, 2003.*, pages 127–134, 2003.

[Kas19] The number of mobile malware attacks doubles in 2018, as cybercriminals sharpen their distribution strategies. 03 2019. Last accessed: 2020-02-19.

[Ken15] Ken Dunham, Shane Hartman, Jose Andre Morales, Manu Quintans, Tim Strazzere. *Android Malware and Analysis.* CRC Press, 6000 Broken Sound Parkway NW, Suite 300, Roca Raton, FL 33487-2742, 1 edition, 01 2015.

[KLLT16] N Kiss, J.-F Lalande, Mourad Leslous, and Valérie Tong. Kharon dataset: Android malware under a microscope. 05 2016.

[LAG+17] A. H. Lashkari, A. F. A.Kadir, H. Gonzalez, K. F. Mbah, and A. A. Ghorbani. Towards a network-based framework for android malware detection and characterization. In *2017 15th Annual Conference on Privacy, Security and Trust (PST)*, pages 233–23309, 08 2017.

[Li,17] Li, Li and Gao, Jun and Hurier, Médéric and Kong, Pingfan and Bissyandé, Tegawendé and Bartel, Alexandre and Klein, Jacques and Le Traon, Yves. Androzoo++: Collecting millions of android apps and their metadata for the research community. 09 2017.

[LNW+14] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. v. d. Veen, and C. Platzer. Andrubis – 1,000,000 apps later: A view on current android malware behaviors. In *2014 Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, pages 3–17, 2014.

[LTLG18] J. Lalande, V. V. T. Tong, M. Leslous, and P. Graux. Challenges for reliable and large scale evaluation of android malware analysis. In *2018 International Conference on High Performance Computing Simulation (HPCS)*, pages 1068–1070, 2018.

[MA14] Aziz Mohaisen and Omar Alrawi. Av-meter: An evaluation of antivirus scans and labels. In Sven Dietrich, editor, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 112–131, Cham, 2014. Springer International Publishing.

[MAC+15] Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu, and Giorgio Giacinto. Stealth attacks: An extended insight into the obfuscation effects on android malware. *Computers & Security*, 51, 03 2015.

[Mal20] 2020 State of Malware Report. https://resources.malwarebytes.com/files/2020/02/2020_State-of-Malware-Report.pdf, 02 2020. Last accessed: 2020-06-04.

[MALM14] Aziz Mohaisen, Omar Alrawi, Matt Larson, and Danny McPherson. Towards a methodical evaluation of antivirus scans and labels. In Yongdae Kim, Heejo Lee, and Adrian Perrig, editors, *Information Security Applications*, pages 231–241, Cham, 2014. Springer International Publishing.

[MD12] Steve Mansfield-Devine. Android malware and mitigations. *Network Security*, 2012(11):12 – 20, 2012.

[MGLCC18] Alejandro Martín García, Raul Lara-Cabrera, and David Camacho. A new tool for static and dynamic android malware analysis. pages 509–516, 09 2018.

[MLCC19] 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. *Information Fusion*, 52:128 – 142, 2019.

[MM00] G. McGraw and G. Morrisett. Attacking malicious code: A report to the infosec research council. *IEEE Software*, 17(5):33–41, Sep. 2000.

[MMF19] Jelena Milosevic, Miroslaw Malek, and Alberto Ferrante. Time, accuracy and power consumption tradeoff in mobile malware detection systems. *Computers & Security*, 82:314 – 328, 2019.

[MMS17] Fabio Martinelli, Francesco Mercaldo, and Andrea Saracino. Bridemaid: An hybrid tool for accurate detection of android malware. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, page 899–901, New York, NY, USA, 2017. Association for Computing Machinery.

[RDG+12] C. Rossow, C. J. Dietrich, C. Grier, C. Kreibich, V. Paxson, N. Pohlmann, H. Bos, and M. v. Steen. Prudent practices for designing malware experiments: Status quo and outlook. In *2012 IEEE Symposium on Security and Privacy*, pages 65–79, 2012.

[Rey20] Roy Reynolds. The four biggest malware threats to uk businesses. *Network Security*, 2020(3):6 – 8, 2020.

[SD19] Raj Samani and Gary Davis. McAfee Mobile Threat Report Q1, 2019. https://www.mcafee.com/enterprise/en-us/assets/reports/rp-mobile-threat-report-2019.pdf, 2019. Last accessed: 2020-06-04.

[SDK19] <uses-sdk>. https://developer.android.com/guide/topics/manifest/uses-sdk-element, 12 2019. Last accessed: 2020-02-27.

[Sip20] Sipior, Janice C., Lombardi, Danielle R., Rusinko, Cathy A., and Dannemiller, Steven. Cyberespionage goes mobile: Fasttrans company attacked. *Communications of the Association for Information Systems*, 46:316–330, 03 2020.

[SMJ16] Vlasta Stavova, Vashek Matyas, and Mike Just. On the impact of warning interfaces for enabling the detection of potentially unwanted applications. In *Proceedings of The 1st European Workshop on Usable Security*. Internet Society, July 2016. DOI does not work 23/8/2016; 1st European Workshop on Usable Security, EuroUSEC 2016 ; Conference date: 18-07-2016 Through 18-07-2016.

[SRKC16] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. Avclass: A tool for massive malware labeling. In Fabian Monrose, Marc Dacier, Gregory Blanc, and Joaquin Garcia-Alfaro, editors, *Research in Attacks, Intrusions, and Defenses*, pages 230–253, Cham, 2016. Springer International Publishing.

[SSDM18] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli. Madam: Effective and efficient behavior-based android malware detection and prevention. *IEEE Transactions on Dependable and Secure Computing*, 15(1):83–97, Jan 2018.

[Sta19] Mobile operating system market share worldwide. https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/, 11 2019. Last accessed: 2020-04-01.

[SZ03] Ed Skoudis and Lenny Zeltser. *Malware: Fighting Malicious Code*. Prentice Hall PTR, USA, 2003.

[TBA12] W. B. Tesfay, T. Booth, and K. Andersson. Reputation based security model for android applications. In *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 896–901, 06 2012.

[TFA+17] Kimberly Tam, Ali Feizollah, Nor Anuar, Rosli Salleh, and Lorenzo Cavallaro. The evolution of android malware and android analysis techniques. *ACM Computing Surveys*, 49:1–41, 01 2017.

[TY17] Fei Tong and Zheng Yan. A hybrid approach of mobile malware detection in android. *Journal of Parallel and Distributed Computing*, 103:22 – 31, 2017. Special Issue on Scalable Cyber-Physical Systems.

[Wil14] William Enck, Peter Gilbert Duke, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. 06 2014.

[WL16] Michelle Wong and David Lie. Intellidroid: A targeted input generator for the dynamic analysis of android malware. 01 2016.

[WLR+17] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. Deep ground truth analysis of current android malware. In Michalis Polychronakis and Michael Meier, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 252–276, Cham, 2017. Springer International Publishing.

[ZJ12] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *2012 IEEE Symposium on Security and Privacy*, pages 95–109, 05 2012.

[ZRN10] Vijay Seshadri Zulfikar Ramzan and Carey Nachenberg. Reputation based security an analysis of real world effectiveness, 12 2010.

Chapter A

Data collection and construction

A.1 andro_zoo_download

```
# Input: a set of hashes
# Output: apks

import requests
import csv
import os.path
import common
import sys

zoo_url = 'https://androzoo.uni.lu/api/download'
malware_path = '/home/inarb/exp1/malware/'
benign_path = '/home/inarb/exp1/benign/'

zoo_api_key = common.zoo_api_key

apks_to_download_hashes = sys.argv[-1]

def get_apk_from_zoo(sha256):
    file_to_write = local_apk_path + sha256 + '.apk'
    if not os.path.exists(file_to_write):
        params = (
            ('apikey', zoo_api_key),
            ('sha256', sha256))
        response = requests.get(zoo_url, params=params)
        if "error" in str(response.content):
            print("error")
            with open('exp2_shas_not_found.csv', 'a') as not_found:
                writer = csv.writer(not_found, delimiter=',')
                writer.writerow([sha256])

                not_found.close()
```

```
        else:
            open(file_to_write, 'wb').write(response.content)


def main():
    with open(apks_to_download_hashes) as apks_to_download:
        csv_reader = csv.reader(apks_to_download)
        for sha256 in csv_reader:
            print("Downloading.. " + sha256[0])
            get_apk_from_zoo(sha256[0])


if __name__ == "__main__":
    main()
```

A.2 update_detections_from_vt.py

```
import requests
import common
import csv

virus_total_url = 'https://www.virustotal.com/vtapi/v2/file/report'

virus_total_api_key = common.virus_total_api_key


def get_detections_from_virus_total(sha256):
    count_detects = 0

    params = {'apikey': virus_total_api_key, 'resource': sha256}
    response = requests.get(virus_total_url, params=params).json()

    if response['response_code'] == 0:
        return None

    else:
        scans = response['scans']

        for scanner in scans:
            detected = scans[scanner]['detected']
            if detected:
                count_detects += 1

        return count_detects


def main():
```

```
    samples_from_androzoo = '/home/inarb/benign.csv'

    reader = csv.reader(open(samples_from_androzoo))
    lines = list(reader)

    for line in lines:
        sha256 = line[0]

        updated_detections = get_detections_from_virus_total(sha256)
        line[7] = updated_detections

    writer = csv.writer(open('/home/inarb/benign_updated.csv', 'w'))
    writer.writerows(lines)


if __name__ == "__main__":
    main()
```

A.3 get_reports_from_vt.py

```
import requests
import json
import common
import os

virus_total_url = 'https://www.virustotal.com/vtapi/v2/file/report'

virus_total_api_key = common.virus_total_api_key


def get_full_reports(sha256):
    count_detects = 0

    if not os.path.exists(f'virus_total_reports_all15/{sha256}.json'):
        try:
            params = {'apikey': virus_total_api_key, 'resource': sha256
    }
            response = requests.get(virus_total_url, params=params).
    json()
            scans = response['scans']

            for scanner in scans:
                detected = scans[scanner]['detected']
                if detected:
                    count_detects += 1

            if count_detects >= 15:
```

```
                with open(f'virus_total_reports_all15/{sha256}.json', 'w') as report_file:
                    json.dump(response, report_file)
        except Exception as e:
            print(e)
            pass


def main():
    hash_file = '15_pluss_detections_shas.txt'

    with open(hash_file, 'r') as hash_file:
        for sha256 in hash_file:
            get_full_reports(sha256)


if __name__ == "__main__":
    main()
```

Chapter B
Selection of tools

B.1 filter_min_sdk.py

```
import os
import androguard.core.bytecodes.apk as apk
import common

max_sdk_AndroPyTool = 16


def remove_files_with_sdks_greater_16():
    for file in os.listdir(common.APK_folder):
        file_path = common.APK_folder + file

        app = apk.APK(file_path)

        min_sdk = app.get_min_sdk_version()

        if min_sdk is None:
            min_sdk = 999

        if int(min_sdk) > 16:
            os.remove(file_path)
            print("[-] Removed file " + file )


if __name__ == "__main__":
    remove_files_with_sdks_greater_16()
```

B.2 automate_mobsf_dynamic.py

```
import requests
from selenium import webdriver
```

```
from bs4 import BeautifulSoup
import time

MobSF_base_URL = 'http://localhost:8000/'
dynamic_analysis_page = requests.get(MobSF_base_URL + 'dynamic_analysis
    /')

soup = BeautifulSoup(dynamic_analysis_page.content, 'html.parser')

uploaded_apps = soup.find_all('a', class_="btn btn-success")

driver = webdriver.Firefox()

for app in uploaded_apps:
    link = app['href']
    start_dynamic_page = driver.get(MobSF_base_URL + link.replace('../'
    , ''))

    checkbox_enumerate_loaded_classes = driver.find_element_by_id("
    enum_class")
    checkbox_enumerate_loaded_classes.click()
    checkbox_capture_strings = driver.find_element_by_id("string_catch"
    )
    checkbox_capture_strings.click()
    checkbox_enumerate_class_methods = driver.find_element_by_id("
    enum_methods")
    checkbox_enumerate_class_methods.click()
    checkbox_capture_string_comparisons = driver.find_element_by_id("
    string_compare")
    checkbox_capture_string_comparisons.click()

    start_instrumentation_button = driver.find_element_by_id("
    frida_spawn")
    start_instrumentation_button.click()

    print("starting dynamic analysis...")
    time.sleep(60)  # Run dynamic analysis for 60 sec

    generate_report_button = driver.find_element_by_id("stop")
    generate_report_button.click()
    print("[+] Generated report for " + link.replace('../
    android_dynamic/?', ''))
```

Chapter C
Final toolchain

C.1 file_events.py

```
1  import os
2  import subprocess
3  import json
4  import csv
5  import time
6  import logging
7  import requests
8  import threading
9  import queue
10
11 from watchdog.observers import Observer
12
13 from androguard.core.bytecodes.apk import APK
14 from androguard.session import Session
15
16 from watchdog.events import FileSystemEventHandler
17
18 import dynamic_analysis
19 import common
20
21 logging.basicConfig(level=os.environ.get("LOGLEVEL", "INFO"))
22
23
24 def get_info(filename): # for debugging
25     p = subprocess.Popen(['file', filename], stdout=subprocess.PIPE)
26     response, err = p.communicate()
27     logging.info(response)
28
29     try:
30         apk = APK(filename)
31
```

```
32          logging.info("APK is signed: {}".format(apk.is_signed()))
33
34          if apk.is_signed():
35              logging.info("APK is signed with: {}".format("both" if apk.
    is_signed_v1() and apk.is_signed_v2()
36                                                       else "v1" if
    apk.is_signed_v1() else "v2"))
37
38      except Exception as exception:
39          logging.error(str(exception))
40          pass
41
42
43  def get_dynamic_analysis_score(file):
44      sample_id = dynamic_analysis.create_new_sample(file)
45      dynamic_analysis.create_new_task_from_sample(sample_id)
46      global_risk_score = dynamic_analysis.get_global_risk_score(
    sample_id)
47      dynamic_analysis.cleanup(sample_id)
48
49      return global_risk_score
50
51
52  def get_static_score(file):
53      threat = None
54
55      p = subprocess.Popen(["/bin/bash", "esetScanner.sh", file], stdout=
    subprocess.PIPE)
56      response, err = p.communicate()
57      response.decode('utf-8')
58      response = str(response)
59
60      if "threat=" in response:
61          response = response.split("threat=")
62          threat = response[1].split(",")[0]
63
64      return threat
65
66
67  def get_reputation_score(sha256):
68      p = subprocess.Popen(["/bin/bash", "get_reputation.sh", sha256],
    stdout=subprocess.PIPE)
69      response, err = p.communicate()
70      response.decode()
71      response_json = json.loads(response)
72
73      reputation_score = response_json['reputation'][sha256]['security'][
    'score']
```

```
74
75      return reputation_score
76
77
78  def get_detections_from_virus_total(sha256):
79      virus_total_url = 'https://www.virustotal.com/vtapi/v2/file/report'
80      count_detects = 0
81
82      params = {'apikey': common.virus_total_api_key, 'resource': sha256}
83      response = requests.get(virus_total_url, params=params).json()
84      scans = response['scans']
85
86      for scanner in scans:
87          detected = scans[scanner]['detected']
88          if detected:
89              count_detects += 1
90
91      return count_detects
92
93
94  def process_load_queue(__queue):
95      while True:
96          if not __queue.empty():
97              event = __queue.get()
98              filename = os.path.abspath(event.src_path)
99
100             logging.info("received new file: " + filename)
101             get_info(filename)
102
103             total_start_time = time.time()
104             reputation_time = 0
105             static_time = 0
106             dynamic_time = 0
107
108             try:
109                 s = Session()
110                 sha256 = s.add(filename)
111
112                 try:
113                     start_time = time.time()
114                     reputation_score = get_reputation_score(sha256)
115                     end_time = time.time()
116                     reputation_time = end_time - start_time
117                 except Exception as exception:
118                     logging.error("reputation error")
119                     logging.error(str(exception))
120                     reputation_score = ""
121                     pass
```

```
122
123                 try:
124                     start_time = time.time()
125                     dynamic_score = get_dynamic_analysis_score(filename
    )
126                     end_time = time.time()
127                     dynamic_time = end_time - start_time
128                 except Exception as exception:
129                     logging.error("dynamic error")
130                     logging.error(str(exception))
131                     dynamic_score = ""
132                     pass
133
134                 try:
135                     start_time = time.time()
136                     static_score = get_static_score(file)
137                     end_time = time.time()
138                     static_time = end_time - start_time
139                 except Exception as exception:
140                     logging.error("static error")
141                     logging.error(str(exception))
142                     static_score = ""
143                     pass
144
145                 total_end_time = time.time()
146                 elapsed_time = total_end_time - total_start_time
147
148                 virus_total_detections =
    get_detections_from_virus_total(sha256)
149
150                 with open('first_results_exp3.csv', 'a') as
    results_file:
151                     writer = csv.writer(results_file, delimiter=',')
152                     writer.writerow([filename, sha256, reputation_score
    , reputation_time, static_score, static_time,
153                                      dynamic_score, dynamic_time,
    elapsed_time, virus_total_detections])
154
155                     results_file.close()
156
157                 logging.info(f'Finished sample {sha256}')
158
159             except Exception as exception:
160                 logging.error(str(exception))
161                 pass
162         else:
163             time.sleep(1)
164
```

```
165
166 class NewFilesEventHandler(FileSystemEventHandler):
167     def __init__(self, __queue):
168         super().__init__()
169         self.__queue = __queue
170
171     def on_created(self, event):
172         self.process(event)
173
174     def process(self, event):
175         self.__queue.put(event)
176
177
178 if __name__ == "__main__":
179     src_path = '/home/inarb/listen/'
180
181     watchdog_queue = queue.Queue()
182
183     for i in range(1, 3):
184         worker = threading.Thread(target=process_load_queue, args=(
    watchdog_queue,))
185         worker.setDaemon(True)
186         worker.start()
187
188     event_handler = NewFilesEventHandler(watchdog_queue)
189     observer = Observer()
190     observer.schedule(event_handler, path=src_path)
191     observer.start()
192
193     try:
194         while True:
195             time.sleep(2)
196     except KeyboardInterrupt:
197         observer.stop()
198
199     observer.join()
```

C.2 eset_scanner.sh

```
1 #!/bin/bash
2
3 file=$1
4
5 /opt/eset/esets/sbin/esets_scan --no-quarantine "$file"
```

C.3 get_reputation.sh

```
#!/bin/bash

sha256=$1

curl -k --data '{"sha256_list": ["'"$sha256"'"]}' --cert client.crt --key client.key https://cloudberry-elb-dev-c0f24215e489f3df.elb.eu-west-1.amazonaws.com:8443/v1/hash |
python -m json.tool
```

C.4 dynamic_analysis.py

```
import subprocess
import json
import requests
import time
import common
import logging

requests.packages.urllib3.disable_warnings()

api_key = common.bluecoat_api_key

headers = {
    'X-API-TOKEN': api_key,
}


def create_new_sample(file):
    p = subprocess.Popen(["/bin/bash", "upload_to_bluecoat.sh", file], stdout=subprocess.PIPE)
    response, err = p.communicate()
    response.decode()
    response_json = json.loads(response)
    sample_id = response_json["results"][0]["samples_basic_sample_id"]
    return sample_id


def create_new_task_from_sample(sample_id):
    data = {
        'sample_id': sample_id,
        'env': 'ivm'
    }

    requests.post('https://research01.osl.bluecoat.com/rapi/tasks', headers=headers, data=data, verify=False)
```

```
33
34
35 def get_global_risk_score(sample_id):
36     response = requests.get(f'https://research01.osl.bluecoat.com/rapi/
       samples/{sample_id}/tasks', headers=headers,
37                             verify=False).json()
38
39     state = response['results'][0]['task_state_state']
40     while "CORE_COMPLETE" not in state:
41         logging.info(state)
42         response = requests.get(f'https://research01.osl.bluecoat.com/
       rapi/samples/{sample_id}/tasks', headers=headers,
43                                 verify=False).json()
44         state = response['results'][0]['task_state_state']
45         if "CORE_INTASKQUEUE" in state:
46             time.sleep(2)
47         else:
48             time.sleep(1)
49
50     global_risk_score = response['results'][0]['tasks_global_risk_score
       ']
51     return global_risk_score
52
53
54 def cleanup(sample_id):
55     response = requests.delete(f'https://research01.osl.bluecoat.com/
       rapi/samples/{sample_id}', headers=headers,
56                                verify=False)
57     if response.status_code == 200:
58         logging.info(f'Successfully deleted sample {sample_id} from
       bluecoat')
59     else:
60         logging.warning("Could not delete. Response code: " + str(
       response.status_code))
```

C.5 upload_to_bluecoat.sh

```
1 #!/bin/bash
2
3 source ../api_key.txt
4
5 file=$1
6
7 curl -k -X POST --form upload=@"$file" --form "owner=iinusen7" --form "
      extension=apk" https://research01.osl.bluecoat.com/rapi/samples/
      basic -H "X-API-TOKEN:$bluecoat"
```

C.6 individual_methods_and_pipeline_1.py

```
import csv
import re

malware_results = 'results_done/exp1_with_family.csv'
# sha256, reputation_score, reputation_time, static_verdict,
    static_time, dynamic_score, dynamic_time, family

benign_results = '/' \
                 'results_done/benign_results_exp1_updated.csv'
# sha256, rep_score, rep_time, stat_score, stat_time, dyn_score,
    dyn_time


def reputation_static_dynamic_alone(lines_malware, lines_benign):
    false_positives_rep = 0
    true_positives_rep = 0
    false_negatives_rep = 0
    true_negatives_rep = 0
    total_time_rep = 0.0

    false_positives_static = 0
    true_positives_static = 0
    false_negatives_static = 0
    true_negatives_static = 0
    total_time_static = 0.0

    false_positives_dyn = 0
    true_positives_dyn = 0
    false_negatives_dyn = 0
    true_negatives_dyn = 0
    total_time_dyn = 0.0

    for line in lines_malware:
        family = line[7]
        try:
            reputation_score = int(line[1])
            reputation_time = float(line[2])
        except:
            reputation_score = 0
            reputation_time = 0.0
            pass

        try:
            static_score = line[3]
            static_time = float(line[4])
        except:
```

```
            static_score = ''
            static_time = 0
        try:
            dynamic_score = int(line[5])
            dynamic_time = float(line[6])
        except:
            dynamic_score = 0
            dynamic_time = 0

        total_time_rep += reputation_time

        if reputation_score < 0:
            true_positives_rep += 1
        else:
            false_negatives_rep += 1

        total_time_dyn += dynamic_time

        if dynamic_score > 7:
            true_positives_dyn += 1
        else:
            false_negatives_dyn += 1

        total_time_static += static_time

        if static_score is not None and re.search("[a-zA-Z]",
    static_score):
            verdict = 'malware'
            true_positives_static += 1
        else:
            verdict = 'benign'
            false_negatives_static += 1

        if verdict == 'benign':
            with open(f'/'
                      f'results_done/pipelines_exp1_new/
    AAstatic_cant_detect.csv', 'a') \
                    as results_file:
                writer = csv.writer(results_file, delimiter=',')
                writer.writerow([line[0], verdict, family])


    for line in lines_benign:
        try:
            reputation_score = int(line[1])
            reputation_time = float(line[2])
        except:
            reputation_score = 0
```

```
91              reputation_time = 0.0
92          try:
93              static_score = line[3]
94              static_time = float(line[4])
95          except:
96              static_score = ''
97              static_time = 0
98          try:
99              dynamic_score = int(line[5])
100             dynamic_time = float(line[6])
101         except:
102             dynamic_score = 0
103             dynamic_time = 0
104
105         total_time_rep += reputation_time
106
107         if reputation_score < 0:
108             false_positives_rep += 1
109         else:
110             true_negatives_rep += 1
111
112         total_time_dyn += dynamic_time
113
114         if dynamic_score > 7:
115             false_positives_dyn += 1
116         else:
117             true_negatives_dyn += 1
118
119         total_time_static += static_time
120
121         if static_score is not None and re.search("[a-zA-Z]",
    static_score):
122             false_positives_static += 1
123         else:
124             true_negatives_static += 1
125
126
127     calculate_and_write_results('Reputation ', true_positives_rep,
    false_negatives_rep, false_positives_rep,
128                                 #true_negatives_rep, total_time_rep)
129
130     calculate_and_write_results('Static ', true_positives_static,
    false_negatives_static, false_positives_static,
131                                 true_negatives_static,
    total_time_static)
132
133     calculate_and_write_results('Dynamic ', true_positives_dyn,
    false_negatives_dyn, false_positives_dyn,
```

```
                                true_negatives_dyn, total_time_dyn)


def pipeline_1(lines_malware, lines_benign):
    false_positives = 0
    true_positives = 0
    false_negatives = 0
    true_negatives = 0
    total_time = 0.0


    for line in lines_malware:
        family = line[7]
        verdict = 'malware'
        try:
            reputation_score = int(line[1])
            reputation_time = float(line[2])
        except:
            reputation_score = 0
            reputation_time = 0.0
        try:
            static_score = line[3]
            static_time = float(line[4])
        except:
            static_score = ''
            static_time = 0
        try:
            dynamic_score = int(line[5])
            dynamic_time = float(line[6])
        except:
            dynamic_score = 0
            dynamic_time = 0

        if reputation_score < 0:
            stage = 'reputation'
            true_positives += 1
            total_time += reputation_time
        else:
            if static_score is not None and re.search("[a-zA-Z]",
    static_score):
                stage = 'static'
                true_positives += 1
                total_time += reputation_time + static_time
            else:
                total_time += reputation_time + static_time +
    dynamic_time
                if int(dynamic_score) > 7:
                    stage = 'dynamic'
```

```
                    true_positives += 1
                else:
                    verdict = 'benign'
                    false_negatives += 1

        if verdict == 'benign':
            with open(f'/'
                      f'results_done/pipelines_exp1_new/
    AApipeline1_cant_detect.csv', 'a') \
                    as results_file:
                writer = csv.writer(results_file, delimiter=',')
                writer.writerow([line[0], verdict, family])

        if verdict == 'malware':
            with open(f'/'
                      f'results_done/pipelines_exp1_new/
    AApipeline1_detect_by.csv', 'a') \
                    as results_file:
                writer = csv.writer(results_file, delimiter=',')
                writer.writerow([line[0], verdict, stage, family])

    for line in lines_benign:
        try:
            reputation_score = int(line[1])
            reputation_time = float(line[2])
        except:
            reputation_score = 0
            reputation_time = 0.0
        try:
            static_score = line[3]
            static_time = float(line[4])
        except:
            static_score = ''
            static_time = 0
        try:
            dynamic_score = int(line[5])
            dynamic_time = float(line[6])
        except:
            dynamic_score = 0
            dynamic_time = 0

        if reputation_score < 0:
            false_positives += 1
            total_time += reputation_time
        else:
            if static_score is not None and re.search("[a-zA-Z]",
    static_score):
                false_positives += 1
```

```
225                 total_time += reputation_time + static_time
226             else:
227                 total_time += reputation_time + static_time +
    dynamic_time
228                 if int(dynamic_score) > 7:
229                     false_positives += 1
230                 else:
231                     true_negatives += 1
232 
233     calculate_and_write_results('Pipeline 1', true_positives,
    false_negatives, false_positives, true_negatives,
234                                 total_time)
235 
236 
237 def calculate_and_write_results(method, true_positives, false_negatives
    , false_positives, true_negatives, total_time):
238     total_samples = 2493 + 2748
239     average_exec_time = total_time / total_samples
240     detection_ratio = true_positives / (true_positives +
    false_negatives)
241     accuracy = (true_positives + true_negatives) / total_samples
242     false_positive_ratio = false_positives / (false_positives +
    true_negatives)
243 
244     with open(f'results_done/pipelines_exp1_new/'
245               f'summary.txt', 'a') as summary:
246         summary.write(f'* {method} * \n true_positives: {true_positives
    }, false_negatives: '
247                       f'{false_negatives}, false_positives: {
    false_positives}, true_negatives: {true_negatives}, '
248                       f'DR: {detection_ratio}, accuracy: {accuracy},
    FPR: {false_positive_ratio} '
249                       f'time: {average_exec_time} \n \n')
250 
251 
252 def run():
253     reader_results = csv.reader(open(malware_results))
254     lines_malware = list(reader_results)
255 
256     reader_benign = csv.reader(open(benign_results))
257     lines_benign = list(reader_benign)
258 
259     reputation_static_dynamic_alone(lines_malware, lines_benign)
260 
261     pipeline_1(lines_malware, lines_benign)
262 
263 
264 if __name__ == '__main__':
```

```
265     run()
```

C.7 pipelines_with_limits.py

```
1  import csv
2  import re
3
4  malware_results = 'results_done/exp1_with_family.csv'
5  # sha256, reputation_score, reputation_time, static_verdict,
       static_time, dynamic_score, dynamic_time, family
6
7  benign_results = 'results_done/' \
8                   'benign_results_exp1_updated.csv'
9  # sha256, rep_score, rep_time, stat_score, stat_time, dyn_score,
       dyn_time
10
11
12 def pipeline_2_malware_static_first(lines_results, limit):
13     false_negatives = 0
14     true_positives = 0
15     total_time = 0.0
16     for line in lines_results:
17         family = line[7]
18         try:
19             reputation_score = int(line[1])
20             reputation_time = float(line[2])
21         except:
22             reputation_score = 0
23             reputation_time = 0
24         try:
25             static_score = line[3]
26             static_time = float(line[4])
27         except:
28             static_score = ''
29             static_time = 0
30         try:
31             dynamic_score = int(line[5])
32             dynamic_time = float(line[6])
33         except:
34             dynamic_score = 0
35             dynamic_time = 0
36
37         if static_score is not None and re.search("[a-zA-Z]",
       static_score):
38             verdict = 'malware'
39             stage = 'static'
40             true_positives += 1
```

```
41              total_time += static_time
42          else:
43              if reputation_score < 0:
44                  verdict = 'malware'
45                  stage = 'reputation'
46                  true_positives += 1
47                  total_time += reputation_time + static_time
48              elif reputation_score >= limit:
49                  verdict = 'benign'
50                  stage = 'reputation'
51                  false_negatives += 1
52              else:
53                  total_time += reputation_time + static_time +
    dynamic_time
54                  if int(dynamic_score) > 7:
55                      verdict = 'malware'
56                      stage = 'dynamic'
57                      true_positives += 1
58                  else:
59                      verdict = 'benign'
60                      stage = 'dynamic'
61                      false_negatives += 1
62
63          with open(f'/'
64                    f'results_done/pipelines_exp1_new/pipeline2_{limit}
    _malware_exp1_static_first.csv', 'a') \
65                  as results_file:
66              writer = csv.writer(results_file, delimiter=',')
67              writer.writerow([line[0], verdict, stage, family])
68
69              results_file.close()
70
71      return true_positives, false_negatives, total_time
72
73
74  def pipeline_2_benign_static_first(lines_results, limit):
75      false_positives = 0
76      true_negatives = 0
77      total_time = 0.0
78      for line in lines_results:
79          family = line[7]
80          try:
81              reputation_score = int(line[1])
82              reputation_time = float(line[2])
83          except:
84              reputation_score = 0
85              reputation_time = 0
86          try:
```

```
87              static_score = line[3]
88              static_time = float(line[4])
89          except:
90              static_score = ''
91              static_time = 0
92          try:
93              dynamic_score = int(line[5])
94              dynamic_time = float(line[6])
95          except:
96              dynamic_score = 0
97              dynamic_time = 0
98
99          if static_score is not None and re.search("[a-zA-Z]",
    static_score):
100             verdict = 'malware'
101             stage = 'static'
102             false_positives += 1
103             total_time += static_time
104         else:
105             if reputation_score < 0:
106                 verdict = 'malware'
107                 stage = 'reputation'
108                 false_positives += 1
109                 total_time += reputation_time + static_time
110             elif reputation_score >= limit:
111                 verdict = 'benign'
112                 stage = 'reputation'
113                 true_negatives += 1
114             else:
115                 total_time += reputation_time + static_time +
    dynamic_time
116                 if int(dynamic_score) > 7:
117                     verdict = 'malware'
118                     stage = 'dynamic'
119                     false_positives += 1
120                 else:
121                     verdict = 'benign'
122                     stage = 'dynamic'
123                     true_negatives += 1
124
125         with open(f'/'
126                   f'results_done/pipelines_exp1_new/pipeline2_{limit}
    _benign_exp1_static_first.csv', 'a') \
127                 as results_file:
128             writer = csv.writer(results_file, delimiter=',')
129             writer.writerow([line[0], verdict, stage, family])
130
131             results_file.close()
```

```
132
133     return false_positives, true_negatives, total_time
134
135
136 def pipeline_2_malware_reputation_first(lines_results, limit):
137     false_negatives = 0
138     true_positives = 0
139     total_time = 0.0
140     for line in lines_results:
141         family = line[7]
142         try:
143             reputation_score = int(line[1])
144             reputation_time = float(line[2])
145         except:
146             reputation_score = 0
147             reputation_time = 0
148         try:
149             static_score = line[3]
150             static_time = float(line[4])
151         except:
152             static_score = ''
153             static_time = 0
154         try:
155             dynamic_score = int(line[5])
156             dynamic_time = float(line[6])
157         except:
158             dynamic_score = 0
159             dynamic_time = 0
160
161         if reputation_score < 0:
162             verdict = 'malware'
163             stage = 'reputation'
164             true_positives += 1
165             total_time += reputation_time + static_time
166         elif reputation_score >= limit:
167             verdict = 'benign'
168             stage = 'reputation'
169             false_negatives += 1
170         else:
171             if static_score is not None and re.search("[a-zA-Z]",
    static_score):
172                 verdict = 'malware'
173                 stage = 'static'
174                 true_positives += 1
175                 total_time += static_time
176             else:
177                 total_time += reputation_time + static_time +
    dynamic_time
```

```
                if int(dynamic_score) > 7:
                    verdict = 'malware'
                    stage = 'dynamic'
                    true_positives += 1
                else:
                    verdict = 'benign'
                    stage = 'dynamic'
                    false_negatives += 1

        with open(f'/'
                  f'results_done/pipelines_exp1_new/pipeline2_{limit}
    _malware_exp1_reputation_first.csv', 'a') \
                as results_file:
            writer = csv.writer(results_file, delimiter=',')
            writer.writerow([line[0], verdict, stage, family])
            results_file.close()

    return true_positives, false_negatives, total_time


def pipeline_2_benign_reputation_first(lines_results, limit):
    false_positives = 0
    true_positives = 0
    total_time = 0.0
    for line in lines_results:
        try:
            reputation_score = int(line[1])
            reputation_time = float(line[2])
        except:
            reputation_score = 0
            reputation_time = 0
        try:
            static_score = line[3]
            static_time = float(line[4])
        except:
            static_score = ''
            static_time = 0
        try:
            dynamic_score = int(line[5])
            dynamic_time = float(line[6])
        except:
            dynamic_score = 0
            dynamic_time = 0

        if reputation_score < 0:
            verdict = 'malware'
            stage = 'reputation'
            false_positives += 1
```

```
225             total_time += reputation_time + static_time
226         elif reputation_score >= limit:
227             verdict = 'benign'
228             stage = 'reputation'
229             true_positives += 1
230         else:
231             if static_score is not None and re.search("[a-zA-Z]",
    static_score):
232                 verdict = 'malware'
233                 stage = 'static'
234                 false_positives += 1
235                 total_time += static_time
236             else:
237                 total_time += reputation_time + static_time +
    dynamic_time
238                 if int(dynamic_score) > 7:
239                     verdict = 'malware'
240                     stage = 'dynamic'
241                     false_positives += 1
242                 else:
243                     verdict = 'benign'
244                     stage = 'dynamic'
245                     true_positives += 1
246
247         with open(f'/'
248                   f'results_done/pipelines_exp1_new/pipeline2_{limit}
    _benign_exp1_reputation_first.csv', 'a') \
249                 as results_file:
250             writer = csv.writer(results_file, delimiter=',')
251             writer.writerow([line[0], verdict, stage])
252             results_file.close()
253
254     return false_positives, true_positives, total_time
255
256
257 def run_pipeline_2():
258     reader_results = csv.reader(open(malware_results))
259     lines_results = list(reader_results)
260
261     reader_benign = csv.reader(open(benign_results))
262     lines_benign = list(reader_benign)
263
264     total_samples = 2493 + 2748
265
266     for limit in range(10, 120, 10):
267         results_reputation_first_malware =
    pipeline_2_malware_reputation_first(lines_results, limit)
```

```
268         results_reputation_first_benign =
    pipeline_2_benign_reputation_first(lines_benign, limit)
269         true_positives = results_reputation_first_malware[0]
270         false_negatives = results_reputation_first_malware[1]
271         false_positives = results_reputation_first_benign[0]
272         true_negatives = results_reputation_first_benign[1]
273         malware_time = results_reputation_first_malware[2]
274         benign_time = results_reputation_first_benign[2]
275
276         average_exec_time = (malware_time + benign_time) /
    total_samples
277         detection_ratio = true_positives / (true_positives +
    false_negatives)
278         accuracy = (true_positives + true_negatives) / total_samples
279         false_positive_ratio = false_positives / (false_positives +
    true_negatives)
280
281         with open(f'/'
282                   f'results_done/pipelines_exp1_new/summary.txt', 'a')
    as summary:
283             summary.write(f'*Reputation first* \n Limit: {limit},
    true_positives: {true_positives}, false_negatives: '
284                           f'{false_negatives}, false_positives: {
    false_positives}, true_negatives: {true_negatives}, '
285                           f'DR: {detection_ratio}, accuracy: {accuracy
    }, FPR: {false_positive_ratio} '
286                           f'time: {average_exec_time} \n \n')
287
288         results_static_first_malware = pipeline_2_malware_static_first(
    lines_results, limit)
289         results_static_first_benign = pipeline_2_benign_static_first(
    lines_benign, limit)
290         true_positives = results_static_first_malware[0]
291         false_negatives = results_static_first_malware[1]
292         false_positives = results_static_first_benign[0]
293         true_negatives = results_static_first_benign[1]
294         malware_time = results_static_first_malware[2]
295         benign_time = results_static_first_benign[2]
296
297         average_exec_time = (malware_time + benign_time) /
    total_samples
298         detection_ratio = true_positives / (true_positives +
    false_negatives)
299         accuracy = (true_positives + true_negatives) / total_samples
300         false_positive_ratio = false_positives / (false_positives +
    true_negatives)
301
302         with open(f'/'
```

```
303                     f'results_done/pipelines_exp1_new/summary.txt', 'a')
    as summary:
304                 summary.write(f'*Static first* \n Limit: {limit},
    true_positives: {true_positives}, false_negatives: '
305                               f'{false_negatives}, false_positives: {
    false_positives}, true_negatives: {true_negatives}, '
306                               f'DR: {detection_ratio}, accuracy: {accuracy
    }, FPR: {false_positive_ratio} '
307                               f'time: {average_exec_time} \n \n')
308
309
310 if __name__ == '__main__':
311     run_pipeline_2()
```

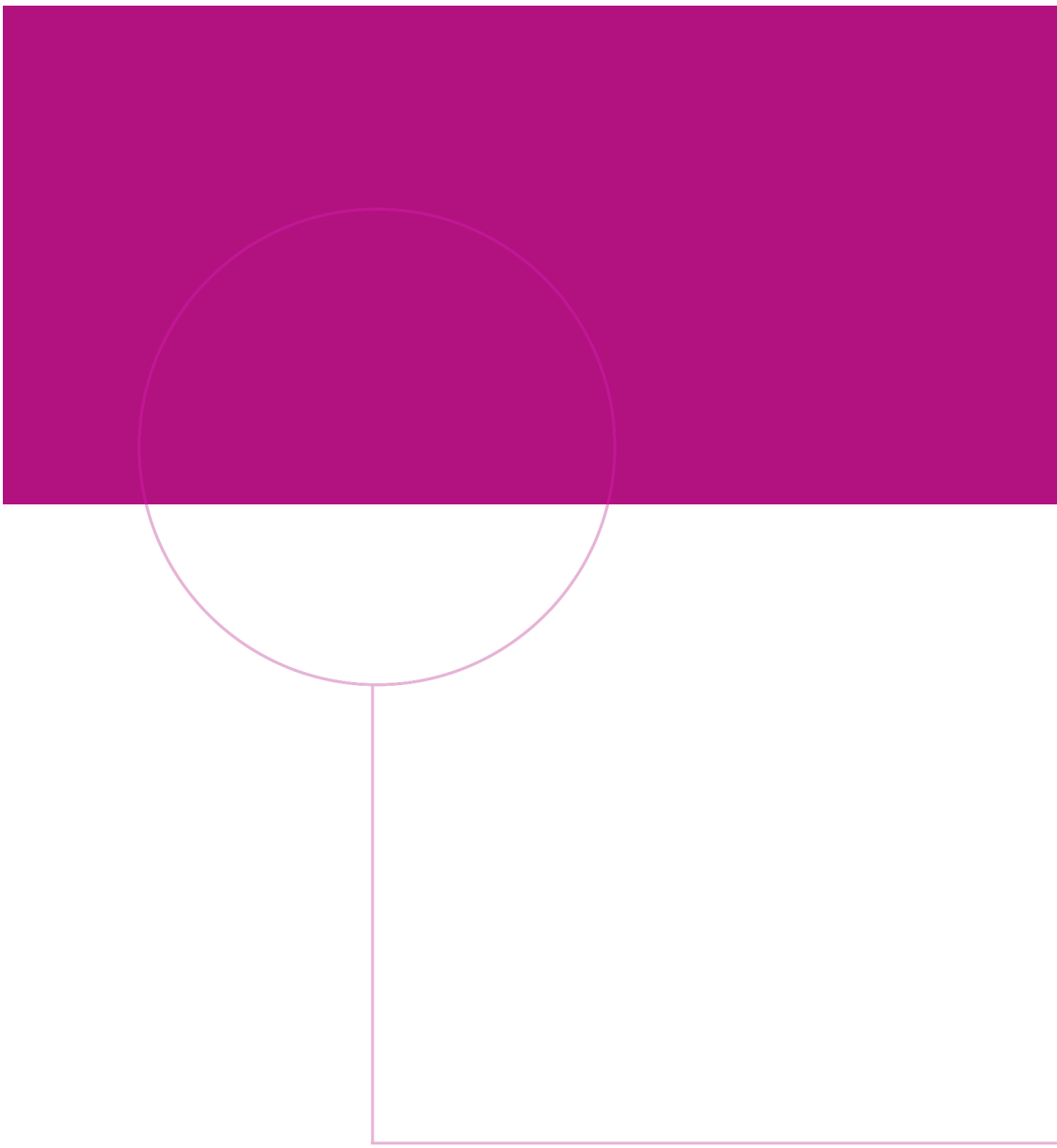
Chapter D
Results

Table D.1 gives the complete results that were explained in Section 6.1.2.

Table D.1: Results for different score limits returned from the reputation-based analysis. We have included results from when the reputation engine is placed first, and when the static analysis is placed first.

Limit	Accuracy	Detection ratio	False positive ratio	Time (s)
Reputation-based analysis first				
10	0.9658	0.9330	4.367×10^{-3}	143.1
20	0.9664	0.9358	5.822×10^{-3}	145.1
30	0.9727	0.9495	6.186×10^{-3}	151.6
40	0.9727	0.9495	6.186×10^{-3}	151.6
50	0.9775	0.9603	6.914×10^{-3}	177.8
60	0.9815	0.9687	6.914×10^{-3}	191.2
70	0.9836	0.9735	7.278×10^{-3}	192.9
80	0.9836	0.9735	7.278×10^{-3}	192.9
90	0.9838	0.9747	8.006×10^{-3}	202.3
100	0.9845	0.9767	8.370×10^{-3}	212.8
110	0.9861	0.9808	9.098×10^{-3}	223.0
Static analysis first				
10	0.9819	0.9667	4.367×10^{-3}	142.9
20	0.9811	0.9667	5.822×10^{-3}	144.9
30	0.9811	0.9671	6.186×10^{-3}	151.3
40	0.9811	0.9671	6.186×10^{-3}	151.3
50	0.9859	0.9779	6.914×10^{-3}	177.6

60	0.9859	0.9779	6.914×10^{-3}	190.9
70	0.9859	0.9783	7.278×10^{-3}	192.7
80	0.9859	0.9783	7.278×10^{-3}	192.7
90	0.9854	0.9783	8.006×10^{-3}	202.0
100	0.9853	0.9783	8.370×10^{-3}	212.5
110	0.9861	0.9808	9.098×10^{-3}	222.7



NTNU

Kunnskap for en bedre verden